Systems and Methods for Measuring and Improving End-User Application Performance on Mobile Devices

by

Ashkan Nikravesh

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Doctoral Committee:

Professor Z. Morley Mao, Chair
Professor Jason N. Flinn
Associate Professor Harsha V. Madhyastha
Assistant Professor Neda Masoud

Ashkan Nikravesh

ashnik@umich.edu

ORCID iD: 0000-0001-9562-5481

To my parents and my sister

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my advisor Professor Morley Mao for her constant presence, continuous support, and constructive feedback throughout my Ph.D. None of this work would have been possible without her help and patience. It has been a true privilege to have her as my advisor.

I would also like to thank my thesis committee: Professor Jason Flinn, Professor Harsha Madhyastha, and Professor Neda Masoud for the helpful discussions and their constructive comments. I am also lucky to have had the opportunity to work with Professor David Choffnes, Professor Feng Qian, and Professor Harsha Madhyastha. They have been instrumental in the completion of this dissertation.

I am very glad that I have had the opportunity to work with Eric Osterweil, my mentor at Verisign. I am very grateful for his support and guidance during my internship. I would also like to thank Shubho Sen, Professor Geoffrey Challen, and Scott Haseley for the fruitful discussions we had in the meetings.

During my Ph.D., I was fortunate to have great graduate student collaborators, including Shichang Xu, Sanae Rosen, David Ke Hong, and Hongyi Yao among many others. I have truly enjoyed working with them. I would also like to thank the rest of RobustNet research group members for many great conversations we had and projects that we did together: Qi Alfred Chen, Yihua Guo, Mehrdad Moradi, Yuru Shao, Yunhan Jia, Xiao Xu.

I am also very grateful to my dear friends in Ann Arbor who have helped me whenever I need them: Mahdi Aghadjani, Morteza Sheikhsofla, Hamed Yousefi, Amir Rahmati, and Alireza Goshtasbi.

Above all, I would like to thank my parents and my sister for their constant and endless source of encouragement, love, support, guidance, and care. I owe all my accomplishments to them.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES ix

LIST OF TABLES xii

ABSTRACT xiii

CHAPTER

I. Introduction 1

1.1 Contributions 4
1.1.1 Mobile Network Performance from User Devices: A Longitudinal, Multidimensional Analysis 4
1.1.2 *Mobilyzer*: An Open Platform for Controllable Mobile Network Measurements 5
1.1.3 An In-depth Understanding of Multipath TCP on Mobile Devices: Measurement and System Design 5
1.1.4 Design and Implementation of an HTTP-based Multipath Solution for Mobile Devices 6
1.1.5 QoE Inference and Improvement Without End-Host Control 7
1.2 Organization 8

II. Related Work 9

2.1 Building Systems for Measuring Mobile Performance 9
2.2 Studies on Mobile Multipath 11
2.3 Studies on QoE Inference 12

III. Mobile Network Performance from User Devices: A Longitudinal, Multidimensional Analysis . . . 15

3.1 Introduction . . . 15
3.2 Methodology and Dataset . . . 16
3.3 Data Analysis . . . 18
3.3.1 Performance across carriers . . . 18
3.3.2 Performance across different locations . . . 20
3.3.3 Performance over time . . . 22
3.3.4 Performance Degradation: Root Causes . . . 24
3.4 Conclusion . . . 26

IV. Mobilyzer: An Open Platform for Controllable Mobile Network Measurements . . . 28

4.1 Introduction . . . 28
4.2 Goals . . . 32
4.2.1 Standard, Easy-to-use Measurements . . . 32
4.2.2 Measurement Isolation . . . 33
4.2.3 Global Coordination . . . 33
4.2.4 Incentives for Adoption . . . 33
4.2.5 Nongoals . . . 35
4.3 Design and Implementation . . . 36
4.3.1 Overview . . . 36
4.3.2 Measurement Library and API . . . 38
4.3.3 Local Measurement Scheduler . . . 41
4.3.4 Global Manager . . . 44
4.3.5 Security . . . 46
4.4 Evaluation . . . 47
4.4.1 Deployment Experience . . . 47
4.4.2 Measurement Isolation . . . 48
4.4.3 Microbenchmarks . . . 49
4.4.4 Server Scheduling . . . 56
4.5 New Measurements Enabled . . . 58
4.5.1 Page Load Time Measurement . . . 59
4.5.2 Video QoE Measurement . . . 67
4.6 Conclusion . . . 70

V. An In-depth Understanding of Mobile Multipath: Measurement and System Design . . . 71

5.1 Introduction . . . 71
5.2 Measurement Methodology . . . 74
5.2.1 Multipath Configuration For User Trial . . . 76
5.2.2 User Trial Data Collection . . . 76

5.3 Measurement Results . . . 78
5.3.1 Passive Measurements of User Study . . . 78
5.3.2 Active Measurements of User Study . . . 83
5.3.3 Other Applications over MPTCP: Voice-over-IP and Instant Messengers . . . 85
5.3.4 Interplay between Multipath and CDN . . . 87
5.4 MPFlex: A Flexible Architecture For Mobile Multipath . . . 91
5.4.1 The MPFlex Architecture . . . 91
5.4.2 MPFlex Design and Implementation . . . 93
5.5 Evaluation of MPFlex . . . 96
5.5.1 File Download . . . 97
5.5.2 Web Browsing . . . 97
5.5.3 Applying Multipath Policies . . . 98
5.5.4 Impact of Proxy Location . . . 99
5.5.5 System Overhead . . . 100
5.6 Concluding Remarks . . . 100

VI. Design and Implementation of an HTTP-based Multipath Solution for Mobile Devices . . . 102

6.1 Introduction . . . 102
6.2 Motivation . . . 104
6.2.1 MPTCP Adoption . . . 105
6.2.2 CDN Server Selection . . . 105
6.2.3 Anycast and Load Balancing . . . 111
6.2.4 Flexible Transport Protocol Support . . . 111
6.3 Design . . . 112
6.3.1 Our Algorithm . . . 116
6.3.2 Implementation Challenges . . . 120
6.4 Application Specific Optimization and Interaction . . . 125
6.4.1 Video Streaming Applications . . . 126
6.4.2 Web based Apps . . . 127
6.5 Implementation . . . 127
6.6 Evaluation . . . 128
6.6.1 Experimental Setup . . . 128
6.6.2 Tail Byte Elimination . . . 128
6.6.3 Single File Download . . . 130
6.6.4 Video Streaming . . . 132
6.7 Conclusion . . . 134

VII. QoE Inference and Improvement Without End-Host Control . . . 135

7.1 Introduction . . . 135
7.2 Motivating Examples . . . 138
7.3 Generating QoS-to-QoE models . . . 141

7.3.1 Recording and Identifying App Usages 142
7.3.2 Replaying App Usage Traces 145
7.3.3 QoE Measurement 146
7.3.4 QoS-to-QoE Mapping 148
7.3.5 Adaptive Sampling of the QoS Metric Space 152
7.3.6 Evaluation 154
7.3.7 Impact of System-level QoS Metrics 154
7.4 QoEBox: QoE-Aware Traffic Management 155
7.4.1 App Name and Usage Detection 156
7.4.2 QoS Measurement 158
7.4.3 Traffic Shaping Rule Generation 159
7.5 Case Studies 160
7.5.1 Classifying and Prioritizing Different Traffic Types . . . 160
7.5.2 QoE-Aware Bandwidth Allocation 163
7.6 Conclusion 168

VIII. Conclusion 169

BIBLIOGRAPHY 172

LIST OF FIGURES

Figure

3.1 Throughput and latency across access technology and carriers. 19
3.2 Verizon LTE Ping RTT in Different Locations 21
3.3 Time of day pattern of HTTP throughput 23
3.4 Weighted Moving Average Error (Median Ping RTT, $W = 2$) 23
3.5 Performance Degradation in: (a)T-Mobile HSDPA network in Bay Area due to server selection flapping from Bay Area to Seattle (b)T-Mobile HSDPA network in Seattle due to change in ingress point of transit AS between T-Mobile and Google (c) Verizon LTE network in Bay Area. . . 25
3.6 Impact of signal strength on latency, packet loss, and throughput. 26
4.1 *Mobilyzer* architecture (shaded region). Apps (top) include the *Mobilyzer* library, which provides an API to issue network measurements, and a scheduler service (middle) that provides measurement management and isolation. The scheduler communicates with a global manager (bottom) to fetch measurement requests and report measurement results. 32
4.2 Using sequential tasks to build a simple diagnosis task. 41
4.3 Impact of throughput measurements (grouped into time periods) and radio state on measured RTT. Results vary with radio state, and upstream or downstream cross-traffic with varying throughput. *Mobilyzer* provides strict control over these kinds of dependencies. 48
4.4 Scheduling overhead for a *Mobilyzer* measurement task. 50
4.5 IPC overhead for DNS burst sizes. 50
4.6 IPC and schedule overhead, 1 app. 51
4.7 Data usage under different data caps. 54
4.8 CDF of CDN-selected server rank (in terms of latency) compared to ten other CDN servers. WiFi and cellular redirections have similar distributions, so we show only the aggregated result here. 55
4.9 CDF of ping latency difference between the CDN-selected server and the CDN IP with lowest latency. 55
4.10 Navigation timing data from crowdsourced PLT measurements, comprising 80 users during six weeks (2000 measurements per page on average). . 61
4.11 Browser computation time versus PLT and PIT. The fraction of load time due to computation is variable, but often a significant portion. 62

4.12 Computation time of different mobile processors (30 distinct device models) compared with desktop. 63
4.13 Computation time in mobile devices can account for more than half of the PLT. (Desktop was connected to a 3Mbps broadband internet service.) 63
4.14 Comparing BBA with CBA bitrates and throughput for users with throughputs that are higher and lower than the maximum bitrate. 66
4.15 Timeline plots of typical throughputs and bitrate adaptations for BBA and CBA. BBA tends to achieve higher average bitrates than CBA with minimal impact on rebuffering. 66
4.16 BBA leads to lower rebuffering rates compared with CBA, particularly for users with lower throughput. 67
5.1 Distributions of the fractions of payload transmitted over the primary subflow, across all MPTCP flows. 75
5.2 Distributions of DL/UL bytes in a SPTCP/MPTCP flow. 75
5.3 Distributions of the handshake delays of MPTCP secondary subflow. . . . 75
5.4 Average TCP throughput for different flow size groups. 80
5.5 Fraction of data delivered by cellular versus the fraction of energy consumed by cellular for popular apps. 80
5.6 Throughput ratio of MPTCP (WiFi primary) to SPTCP over WiFi for 4,371 back-to-back throughput measurements for different value of LTE signal strength. 81
5.7 Crowd-sourced video streaming measurements (1,500 measurements in total) 84
5.8 QoE and power consumption of VoIP under different SPTCP/MPTCP settings. 85
5.9 Example: CDN sever selection over multipath. 87
5.10 File download time from different CDN servers, averaged over 10 runs. Shaded blue is a "what-if" scenario. 87
5.11 Distributions of path latency differences for 190 out of Alexa top-500 websites. Refer to Figure 5.9 for the four paths. 88
5.12 The MPFlex architecture. 92
5.13 Components within an MPFlex endpoint. 93
5.14 Single file download over MPTCP and MPFlex (best SPTCP results shown only for small downloads). 95
5.15 Transfer many short flows over MPTCP and MPFlex. 96
5.16 Fetch web pages over MPTCP and MPFlex. 96
5.17 Case study: MPTCP applies multipath to all traffic, while MPFlex does that selectively based on user policy. 98
5.18 Performance impact of MPFlex proxy location. 98
6.1 Netflix server selection in the context of MPTCP. 106
6.2 Impact of sub-optimal server selection on the download time of Netflix video data chunks. 106
6.3 Impact of sub-optimal server selection on the performance of MPTCP (emulation). 107
6.4 Crowd-sourced latency measurement of multipath server selection. 109

6.5 Comparing MPTCP performance with constant chunk size scheme with different sizes when downloading 512KB and 1MB files. 112
6.6 Timeline of sending HTTP requests to download data chunks on WiFi and cellular interfaces. 113
6.7 Two iterations of the byte range adjustment. Grayed area shows the downloaded parts of the data chunks. 114
6.8 Performance of MP-HTTP scheduler for downloading a 2MB file on a device connected to 10Mbps WiFi and 5Mbps cellular network. Y-axis shows the byte ranges requested on different interfaces. 115
6.9 Timeline of sending HTTP requests and HTTP/2.0 control frames to download two chunks of data, S1 and S2, on a single connection. 119
6.10 Comparing the performance of the proposed tail byte elimination with canceling a stream using HTTP/2.0 `RST_STREAM` frame. 129
6.11 Compare performance of different schedulers under three bandwidth settings. 129
6.12 Compare performance of different schedulers under real bandwidth profiles. 130
6.13 Compare the video quality of different schedulers under three capped bandwidth profiles. 133
6.14 Compare the video quality of different schedulers under real bandwidth profiles. 133
7.1 Problem of resource management for two scenarios, in which two users using different apps at the same time. 139
7.2 Number of distinct usages for top 100 apps. 144
7.3 Distribution of the frequency of interactions for each usage for six popular apps. 145
7.4 Our experimental setup for generating QoS-to-QoE mappings 146
7.5 Mapping (a) latency and (b) bandwidth to launch time. 148
7.6 Mapping various QoS metrics to frame rate (QoE) for AppRTC and Skype 151
7.7 Mapping bandwidth (QoS metric) to video bitrate (QoE metric) for Youtube, Microsoft Smooth Streaming, and Apple HLS 151
7.8 Sampled QoS values based on Algorithm 2 153
7.9 Compare accuracy of adaptive sampling with regression-based modelings. 153
7.10 QoEBox architecture. 155
7.11 App response delay of five usage types under three scenarios: without concurrent video traffic, with video traffic, and with prioritization. 155
7.12 Frame rate of AppRTC and Skype w/o and with video traffic, and with prioritization. 163
7.13 Video bitrate of 10 users streaming video at random times and duration. . 164
7.14 Simulation results of different number of users streaming video under 100Mbps bandwidth. 164
7.15 Total MOS of 10 users streaming video and downloading a file at the same time. 165

LIST OF TABLES

Table

3.1	Number of Measurement and Carriers for the Network Technologies	18
4.1	Measurement types supported by *Mobilyzer*.	32
4.2	*Mobilyzer* key API functions to support issuing and controlling measurement tasks.	36
4.3	Lines of code (LoC) for open-source measurement apps. By integrating *Mobilyzer* into existing apps, developers can save thousands of lines of code.	48
4.4	Power consumption of *Mobilyzer*: each cell lists the average, then the standard deviation in parentheses.	52
5.1	Summary of the findings and proposed improvement	74
5.2	Statistics of the user study dataset.	77
5.3	Comparison of three multipath proxy solutions.	93
6.1	Results of DNS lookup over WiFi and cellular for each CDN.	109

ABSTRACT

In today's rapidly growing smartphone society, the time users are spending on their smartphones is continuing to grow and mobile applications are becoming the primary medium for providing services and content to users. With such fast paced growth in smartphone usage, cellular carriers and internet service providers continuously upgrade their infrastructure to the latest technologies and expand their capacities to improve the performance and reliability of their network and to satisfy exploding user demand for mobile data. On the other side of the spectrum, content providers and e-commerce companies adopt the latest protocols and techniques to provide smooth and feature-rich user experiences on their applications.

To ensure a good quality of experience, monitoring how applications perform on users' devices is necessary. Often, network and content providers lack such visibility into the end-user application performance. In this dissertation, we demonstrate that *having visibility into the end-user perceived performance, through system design for efficient and coordinated active and passive measurements of end-user application and network performance, is crucial for detecting, diagnosing, and addressing performance problems on mobile devices.* My dissertation consists of three projects to support this statement. First, to provide such continuous monitoring on smartphones with constrained resources that operate in such a highly dynamic mobile environment, we devise efficient, adaptive, and coordinated systems, as a platform, for active and passive measurements of end-user performance. Second, using this platform and other passive data collection techniques, we conduct an in-depth user trial of mobile multipath to understand how Multipath TCP (MPTCP) performs

in practice. Our measurement study reveals several limitations of MPTCP. Based on the insights gained from our measurement study, we propose two different schemes to address the identified limitations of MPTCP. Last, we show how to provide visibility into the end-user application performance for internet providers and in particular home WiFi routers by passively monitoring users' traffic and utilizing per-app models mapping various network quality of service (QoS) metrics to the application performance.

CHAPTER I

Introduction

Mobile data traffic has grown 18-fold over the past five years [8]. As of March 2017, between 2.8 and 2.2 million apps were available for Android and iOS users, respectively [32]. For a single type of service or app, users are able to choose between a wide variety of apps providing the same service, including social media, messaging, news, media and entertainment, and shopping apps.

In such a hyper-competitive app market, app developers try to deliver the most satisfying service to their clients by improving the performance of their app. To do so, they continuously propose, implement, or adopt new transport-layer network protocols and application-layer adaptation techniques that better fit mobile networks with highly dynamic network conditions. For instance, to reduce latency, Google proposed QUIC in 2012 and has been using that ever since. QUIC is by default enabled in the Chrome browser and about half of all requests from Chrome to Google web servers happen over QUIC [38]. Facebook has also built an experimental Zero protocol over TCP based on QUIC's crypto protocol [5], to provide fast and secure data transfers on its app. Apple has been using Multipath TCP (MPTCP) to improve the reliability of its digital assistant (Siri), since 2013.

Along with app developers' efforts, wireless carriers are also heavily invested and engaged in updating their infrastructure and adopting the next generations of mobile communication technology to better accommodate high bandwidth demands of their users. For

instance, AT&T plans to introduce mobile 5G service in a dozen markets by late 2018 [4].

When adopting a new protocol or technology, understanding how it performs on users' devices in this highly dynamic and mobile environment is important. To be able to identify inefficiencies in its design, finding factors impacting performance, and detecting problems causing performance degradation, continuous measurement of performance on the users' devices is essential. Ideally, app developers and internet service providers should be able to collect performance measurement data from any mobile device on any network at any time. With this information, wireless carriers could study the factors impacting performance and detect problems causing degradation, and application developers could tune and improve their service quality. This information is even useful for the users, as they could evaluate the services they are paying for.

In this dissertation, through three main projects, we design systems and techniques to *measure* and *infer* end-user application performance for various types of apps. We characterize the performance of Internet-scale systems and protocols by conducting in-lab and crowd-sourced measurement studies, diagnose their performance bottlenecks, and propose solutions to address them.

In the first project [131, 134], we first study end-to-end performance as seen from mobile devices, using a dataset of scheduled network measurements spanning more than 100 carriers over 17 months. We find that there are significant performance differences across carriers, access technologies, geographic regions, and over time. Further, we find that these variations themselves are not uniform, making network performance difficult to diagnose. Another reason that contributes to the fact that we are unable to find the root cause behind these variations is the lack of coordination and control over the scheduled measurements. Ideally, we would be able to dynamically change measurements' parameters based on the collected results to factor out the impact of each individual element that might cause the observed performance degradation.

Motivated by our findings in this study, we design and implement a platform for con-

ducting *controllable* mobile network measurement experiments, called *Mobilyzer*. *Mobilyzer* provides visibility into the end-user application performance by supporting various types of complex application-layer measurements. It enables performance evaluation of Internet-scale systems (e.g., network protocols, video adaptation schemes, etc.) in the mobile environment through its support of coordinated measurements among large numbers of vantage points and efficient use of scarce resources on mobile devices. Using *Mobilyzer*, we could identify and diagnose cases of CDN inefficiency, characterize and decompose page load time delays for web browsing, and evaluate alternative video bitrate adaptation schemes in the mobile environment. In most of the projects discussed in this thesis, we use *Mobilyzer* as the main platform for conducting large-scale crowd-sourced measurement of mobile network and application performance.

In the second project [132], we augment the platform we developed in the first project with passive measurement techniques to conduct an in-depth measurement study of how mobile Multipath TCP (MPTCP) performs in the wild. In this project, our goal is to quantify the performance benefits MPTCP can offer in real-world settings and to measure its energy footprint on real users' devices. To achieve this goal, we leverage *Mobilyzer* to crowd-source active measurements of application QoE on participants' phones and capture how they perform in multipath settings and under diverse network conditions. Our measurement study reveals several limitations of MPTCP, including its poor interaction with short flows, lack of infrastructural support for multipath policy, and suboptimal CDN server selection. To address these limitations, we propose two solutions: (1) MPFlex, a transport layer solution that employs multiplexing to improve the performance of short flows and enables the apps to enforce their multipath policy; (2) MP-HTTP, an application layer solution which operates on top of HTTP and is easier to adopt. MP-HTTP addresses the sub-optimal CDN server selection and provides comparable performance to the MPTCP scheduler.

Since it is impractical to use active probes to measure application performance at scale

over all the devices and applications, in the last project, we propose a mechanism to infer end-user application performance by passively monitoring users' traffic. In this project, our proposed approach for inferring QoE corresponding to a traffic flow is to rely on models that can map the flow's QoS metrics to the corresponding app's QoE metrics. We generate these model by developing an app-independent tool that is capable of recording and replaying users' interactions for different types of apps under different QoS values. These generated models enable any entity that can observe an app's network traffic, including ISPs and access points, to infer end-user app performance. We demonstrate how the model can be utilized for the purpose of traffic management for wireless access points by designing, implementing, and evaluating two traffic management schemes: latency-sensitive traffic prioritization and a novel bandwidth allocation scheme for bandwidth-intensive apps. We show that for both schemes, we can significantly improve various app performance metrics that reflect user-perceived performance.

1.1 Contributions

The following paragraphs summarize the major findings and contributions of this thesis, organized by the papers.

1.1.1 Mobile Network Performance from User Devices: A Longitudinal, Multidimensional Analysis

In this paper, we took a first look at end-to-end performance as seen from mobile devices, using a dataset of scheduled network measurements spanning more than 100 carriers over 17 months. In particular, we find that all carriers exhibit significant variance in end-to-end performance in terms of latency and throughput. To explain this variance, we investigate geographic and temporal properties of network performance. While we find that these properties account for some differences in performance, importantly we observe that performance is inherently unstable, with some carriers providing relatively more or less

predictable performance. Last, we identify that routing (*i.e.,* inefficient paths) and signal strength are alternative sources of variance in latency and throughput.

1.1.2 *Mobilyzer*: An Open Platform for Controllable Mobile Network Measurements

In this project, we design and build *Mobilyzer*, a unified platform for conducting network measurements in the mobile environment. The main contributions of this paper are:

- Design and implementation of *Mobilyzer*, a platform for conducting mobile network measurement experiments in a principled manner, that includes an API for issuing network measurements using a standard suite of tools, a cloud-based global manager that dynamically deploys measurement experiments to devices according to available resources, device properties and prior measurement results and manages data collection from participating devices.

- Evaluation of *Mobilyzer* in terms of new measurement studies that it enables, using *Mobilyzer*'s support for coordinated measurements among large numbers of vantage points to evaluate the performance of Internet-scale systems. For instance, we use *Mobilyzer* to identify and diagnose cases of CDN inefficiency, characterize and decompose page load time delays for Web browsing, and evaluate alternative video bitrate adaptation schemes in the mobile environment.

1.1.3 An In-depth Understanding of Multipath TCP on Mobile Devices: Measurement and System Design

In this paper, we conduct an in-depth measurement-driven study of multipath over mobile devices, with the goal of providing key knowledge and vital clues for evolving the mobile multipath design. More specifically, our key contributions are as follows:

- We launch an IRB-approved user trial by collecting network traces from real users' smartphones with MPTCP enabled. Leveraging the unique data we collected, we

analyzed MPTCP behaviors"in the wild", including multipath availability, path utilization, handshake latency, TCP throughput, web page load time, video streaming bitrate. From the user study, we find that multipath is widely available and it incurs rather complex interactions with applications due to their diverse traffic patterns and QoE metrics. MPTCP's suboptimal performance often stems from three factors: short flow duration and excessive delay of subflows' handshakes.

- By applying previously validated single-path and multipath radio energy models on real users' network traffic, we find that properly using multipath incurs reasonable and manageable overhead. However, blindly applying MPTCP to all traffic, as is usually done today, increases users' average radio energy by 1.08X.

- We find that under multipath, the state-of-the-art DNS-based server selection often leads to suboptimal server selection. This is attributed to the fact that CDNs DNS infrastructure is unaware of other subflows available to the client. We demonstrate its real world importance by probing Alexa top-500 websites and provide recommendations for multipath-aware server selection.

- To address the identified limitations of MPTCP, we propose a flexible software architecture of mobile multipath, called MPFlex. MPFlex performs transparent multiplexing to improve multipath performance and provides an ideal vantage point for flexibly realizing user-specified multipath policies.

1.1.4 Design and Implementation of an HTTP-based Multipath Solution for Mobile Devices

In this paper, we propose Multipath HTTP (MP-HTTP), a new client side multipath solution with optimized scheduler that operates on top of HTTP. MP-HTTP can be integrated into applications seamlessly, *i.e.,* without any modification to the client and server. They key contributions are:

- We design a client side multipath scheduler on top of HTTP/2.0 that aims to achieve simultaneous chunk completion time on WiFi and cellular interfaces and fully utilize their bandwidth.

- We implement MP-HTTP scheduler for Android devices and compare its performance with MPTCP and other HTTP-based schedulers by conducting extensive measurements in realistic settings. Our results show that MP-HTTP provides comparable performance to MPTCP scheduler and brings significant performance boost compared to other state-of-the-art HTTP- based schedulers.

- As one of the main applications of MP-HTTP, we further optimize its performance for ABR video streaming. We include MP-HTTP into a popular video player framework and show that under realistic bandwidth settings, its performance is close to MPTCP and it outperforms other state-of-the-art HTTP-based schedulers.

1.1.5 QoE Inference and Improvement Without End-Host Control

In this paper, we propose offline generation of per-app models mapping app-independent QoS metrics to app-specific performance metrics. This enables any entity that can passively observe an app's network traffic—including ISPs and access points—to infer the end-user app performance. In particular, our contributions are:

- We generate the models by replaying real users' common interactions in popular interactive apps, two popular video conferencing apps, and three video streaming apps, with significantly different QoE metrics. We identify important combination of QoS values that causes change in QoE of these apps using a new adaptive sampling technique.

- We design and implement QoEBox, a proxy-based QoE-centric traffic management framework. QoEBox maps the traffic belonging to an app to its corresponding QoE model, and invokes custom modules to apply traffic shaping policies.

- We showcase how the models can be utilized for the purpose of traffic management by designing, implementing, and evaluating two QoE-aware traffic managements schemes on WiFi access points: prioritizing latency sensitive traffic and an optimal fair bandwidth allocation for bandwidth intensive apps. We show that for both schemes, we can significantly improve various QoE metrics, including frame rate, app responsiveness, and video bitrate, without modifying the end-host or requiring a very precise model.

1.2 Organization

The rest of this dissertation is organized as follows. We first provide an overview of related work in chapter 2. In chapter 3, we present a measurement study of cellular network performance. Based on the insights gained from this study, we discuss the design and implementation of a platform for conducting network measurement on mobile devices in chapter 4. Using this platform, we study the performance of Multipath TCP on real users' devices in chapter 5. Our measurement results indicate that MPTCP suffers from a few limitations and in chapter 6, we propose a client-side scheduler to address these limitations. Last, in chapter 7, we propose an approach for inferring and improving end-users' application performance by passively monitoring their traffic. This enables various entities that can monitor users' traffic, including wireless carriers and ISPs, to infer end-user application performance without requiring to run any active measurements. Finally, in chapter 8, we conclude with a summary of the contributions of this work and discussion of potential future work.

CHAPTER II

Related Work

The projects described in this dissertation build upon and complement a substantial amount of prior work. In this chapter we discuss the prior work and highlight their limitations that we address. These related studies are organized into three main categories according to the three main projects discussed in this dissertation.

2.1 Building Systems for Measuring Mobile Performance

Many previous studies attempt to improve our visibility into and understanding of mobile network performance. We broadly categorize them in to research platforms, platforms deployed by network operators, and measurement apps.

Existing research platforms. There are many successful testbeds deployed such as PlanetLab [142], RIPE Atlas [154], and BISMark [170], and in our first project (*i.e., Mobilyzer*), we uses M-Lab [120] servers as targets for many measurement tests. These are general-purpose experimentation platforms that require deployment of infrastructure and do not operate in the mobile environment. These systems are insufficient alone for understanding mobile networks because mobile networks generally use firewall/NATs [179].

Our first project shares many goals with Dasu [157], but differ fundamentally in that properly characterizing network performance in mobile environment requires different strategies than in the wired/broadband environment due to scarce resources and measure-

ment interference. Seattle [63] is a general-purpose platform that supports deploying code on Android devices, and it shares many goals with *Mobilyzer*. It does not provide network measurement isolation, but many of its device management and security features can be integrated into *Mobilyzer*.

View from operators. Operators and manufacturers such as AT&T and Alcatel-Lucent have deployed in-network monitoring devices that passively capture a detailed view of network flows traversing their mobile infrastructure [73, 53]. Several studies use these passive measurements to understand network traffic generated by subscriber devices, with important applications to traffic engineering, performance characterization and security [87, 146, 181, 53, 175, 115, 155]. However, this approach does not allow the carrier to understand the performance experienced at the mobile device. For example, when the throughput for a network flow suddenly changes, it is difficult to infer from passive measurements alone whether it is due to wireless congestion, signal strength, and/or simply normal application behavior.

Existing measurement apps. Motivating the need for a mobile measurement library, several academic, governmental and commercial mobile performance measurement tools have been released recently. The FCC released a network measurement app [83] to characterize mobile broadband in the US but the system is closed and data is not publicly available. Several commercial apps measure mobile Internet performance by collecting measurements such as ping latency and throughput [167, 64, 136]. However, because the source code is closed, the methodology is under-specified, and the datasets tightly controlled, it is difficult for the research community and end-users to draw sound conclusions. Further, these tests are generated on-demand by end-users, creating a strong selection bias in the data, as users may be more likely to run tests when they experience problems.

A number of research projects use apps to gather network measurements from hundreds or thousands of mobile devices [164, 97, 143, 112, 162, 161, 123, 126]. Such measurements reveal information not visible from passive in-network measurements, such as radio

state, signal strength and the interaction between content providers and network infrastructure [188, 180, 101, 164]. Further, these measurements reveal information across carriers, allowing researchers to understand different ISP network policies and performance [179]. Such one-off studies provide a useful snapshot of mobile networks, but they do not support longitudinal studies that inform how mobile network performance, reliability and policy changes over time, nor do they support targeted measurements that are required to understand the underlying reasons for observed network behavior.

Our first project is also motivated by other efforts in building measurement libraries, especially on mobile platforms. For example, Insight [139] and AppInsight [152] offer platforms for instrumenting and measuring application performance, rather than network measurements.

2.2 Studies on Mobile Multipath

In this section, we discuss prior work on mobile multipath in four categories.

Performance Characterization of Mobile Multipath. Although the potentials of mobile multipath have been known for a long time [55], it has recently become a hot research topic as fueled by smartphones and MPTCP. Chen *et al.* [70] studied performance of MPTCP over 3G/4G and WiFi. A similar studied was performed by Deng *et al.* [77] to compare the performance between single-path and multipath. Both studies focus on file download using controlled experiments. Han *et al.* [92] investigated how MPTCP helps improve web performance by in-lab experiments. They found SPDY [41] better interacts with multipath compared to HTTP/1.1, due to multiplexing. We make a further step by proposing a flexible transport-layer multiplexing infrastructure for multipath, which provides considerably more benefits than application-layer multiplexing does.

De Coninck *et al.* [72] conducted a measurement study of MPTCP involving 12 mobile users. They focus on transport layer characteristics of MPTCP such as RTT, retransmission, and reinjection. In contrast, our user study focuses not only on MPTCP itself, but also on

the cross-layer interactions. We also combine passive and active measurements in our study to get a thorough understanding of applications' performance and energy utilization in the wild.

Multipath Energy. Some studies also examined the energy aspect of multipath. Nika *et al.* [130] characterized energy and performance of multipath in outdoor environments. Very recently, Lim *et al.* [172] improves MPTCP to make it energy-aware. Peng *et al.* [140] also proposes algorithms that tradeoff throughput performance and energy consumption for MPTCP. We leveraged the multipath power models derived by some of these work, and use them to characterize multipath energy consumption in real-world.

Applications of Mobile Multipath. Besides improving existing applications' performance, MPTCP provides opportunities for enabling new use cases. Mobile Kibbutz [129] is a system allowing nearby users share their links with each other via multiple shorter range wireless links. MSPlayer [71] is a YouTube client that fetches multiple video sources over multipath to improve video experience. Croitoru *et al.* [74] leveraged MPTCP to achieve seamless mobility in WiFi by letting a client connect to multiple APs on the same channel. Our work complements these specific systems by characterizing and improving MPTCP itself. There are other systems dealing with multiple interfaces in general, such as energy-efficient interface selection [141] and utilization of concurrent WiFi APs [165].

2.3 Studies on QoE Inference

There has been a rich body of literature that studies QoS and QoE measurements. In this section, we summarize related work that leverage predictive models to estimating QoE within the network.

QoE Predictive Models. There is a rich body of work that leverages predictive models to estimating video QoE within the network [159, 66, 106]. Schatz *et al.* [159] presented methods to estimate the number of stalling events and their duration for YouTube using network level measurements. Casas *et al.* [66] presented YOUQMON, which can detect

stalling events in YouTube video stream by analyzing the traffic collected in 3G core network, and then map it to MOS.Compared to these video streaming specific methods, our proposed modeling is more broadly applicable to a wide range of apps and QoE metrics. The closest work to ours is Prometheus [51], which estimates app QoE using passive network measurement, and then uses linear regression to map network traffic features to the binary classification of QoE. In contrast, we argue that the QoS-to-QoE mapping may *not* be linear, due to (1) the complex interaction between app protocol and network conditions, and (2) the non-linear relationship between QoS metrics (*e.g.,* bandwidth) and user satisfaction [108]. Moreover, to gather training data for prediction, Prometheus relies on passive measurement of QoS from real mobile devices, while we propose an offline sampling technique that efficiently samples QoS values close to the boundary of different QoE classes. ExBox [67] is a QoE-aware admission control mechanism for WiFi networks that leverages IQX hypothesis model [84] to estimate QoE of incoming flows and then classifies them as admissible or non-admissible.

QoE Aware Traffic Management. Traffic prioritization is a known technique to mitigate in-network bufferbloat [128, 99, 90]. Prioritizing traffic from certain applications using per-class queuing is also recommended by IETF as one of the best practices for active queue management on network devices [56]. As a part of IEEE 802.11e [19] standard, Wireless Multimedia (WMM) service is proposed and supported by commercial WiFi routers to classify and prioritize the traffic of certain types of apps. However, it requires end-device input and classification. In fact WMM service has been always enabled in all our experiments. Bozkurt *et al.* [60] and Martin *et al.* [119] propose traffic management schemes for home networks that rely on users' preference and apps' input, respectively, to prioritize the traffic. Jiang *et al.* [102] propose a network paradigm where apps and network providers can collaborate by exchanging information such as QoE data. In comparison to these approaches, our proposed traffic management framework is designed for unmodified and unaware apps, *i.e.,* it does not need to communicate with end-host and it does not re-

quire users' input. It relies on the QoS-to-QoE models to determine whether a usage traffic is latency sensitive and should be prioritized. Several prior efforts focus on the problem of *bandwidth allocation* for various type of apps, specifically video traffic. FESTIVE [104] is a client side solution which provides a trade-off between fairness, stability and efficiency. Q-Point [79] and QFF [86] are SDN-based approaches to address fairness and maximize aggregated QoE of multiple competing clients simultaneously watching video. In comparison with these approaches that focus on a single type of app, our bandwidth allocation technique can be applied to different types of bandwidth intensive apps. As a practical QoE-aware traffic management framework, we cover all types of apps, and classify and allocate the resources with respect to their QoS-to-QoE model. Bozkurt *et al.* [60] present Contextual Router for home networks, which incorporates users' preferences and existing app QoE models to prioritize traffic and determine bandwidth allocations, respectively. Martin *et al.* [119] propose a traffic prioritization approach for home routers that relies on apps' hint to prioritize certain flows. Jiang *et al.* [102] propose a network paradigm where application and network provides can collaborate by exchanging information such as QoE data. In comparison to these approaches, our proposed traffic management framework is designed for unmodified and unaware apps, *i.e.,* it does not need to communicate with end-host and does not require users' input.

CHAPTER III

Mobile Network Performance from User Devices: A Longitudinal, Multidimensional Analysis

3.1 Introduction

Cellular networks are the fastest growing, most popular and least understood Internet systems. A particularly difficult challenge in this environment is capturing a view of network performance that is representative of conditions at end user devices. A number of factors frustrate our ability to capture this view. For instance, carriers enforce different policies depending on the traffic types or geographic/social characteristics of different locations such as population [164, 173], causing user perceived performance to differ from advertised performance for access technologies. Other environmental factors have a significant impact on performance, including device model [97], mobility [118], network load [173], packet size [105, 116] and MAC-layer scheduling [118].

To account for various factors impacting Internet performance in mobile networks, we need pervasive network monitoring that samples a variety of devices across carriers, access technologies, locations and over time. This work takes a first look at such a view using data collected from controlled measurement experiments in 144 carriers during 17 months, comprising 11 cellular network technologies. We use this data to identify the patterns, trends, anomalies, and evolution of cellular networks' performance.

This study demonstrates that characterizing and understanding the performance in today's cellular networks is far from trivial. We find that all carriers exhibit significant variance in end-to-end performance in terms of latency and throughput. To explain this variance, we investigate geographic and temporal properties of network performance. While we find that these properties account for some differences in performance, importantly we observe that performance is inherently unstable, with some carriers providing relatively more or less predictable performance. Last, we identify alternative sources of variance in performance that include routing and signal strength. An important open question is how to design a measurement platform that allows us to understand reasons behind most observed performance differences.

Our study differs from previous related work in that our study is longitudinal, continuous, pervasive and gathered from mobile devices using controlled experiments. In contrast, some related work [175, 115, 155] passively collected network traffic from cellular network infrastructure, using one month of data or less. These studies tend to be limited to a single carrier, hampering our ability to conduct meaningful comparisons across carriers. Other work collected network performance data at mobile devices [78, 164, 82], but did not use controlled experiments to capture a continuous view of performance.

Roadmap. We describe our methodology and dataset in §3.2, then present our findings regarding network performance across different network technologies, carriers, locations, and times in §3.3.1, §3.3.2, and §3.3.3 respectively. Then we study the root causes for performance degradation in §3.3.4 and conclude in §3.4.

3.2 Methodology and Dataset

In this chapter, we study cellular network performance using a broad longitudinal view of network behavior impacting user-perceived performance. To this end, we consider HTTP GET throughput, round trip time latency from ping, and DNS lookup time as end-to-end performance metrics. In addition to gathering raw performance data, we annotate our mea-

surements with path information gathered from traceroute, the identify of the device's carrier, its cellular network technology, signal strength, location and timestamp.

We focus on performance from mobile devices to Google, a large, popular content provider. We argue that Google is an ideal target for network measurements because it is highly available and well provisioned, making it easier to isolate network performance to cell networks vs. Google's network. Focusing on these measurements, we identify the performance impact of carrier, network technology, location and time. To reason about the root cause behind performance changes, we use path information, DNS mappings and signal strength readings.

Our data is collected by two Android apps using a nearly identical codebase, Speedometer and Mobiperf.[1] Speedometer is an internal Android app developed by Google and deployed on hundreds of volunteer devices, mainly owned by Google employees. As such, the bulk of our dataset[2] is biased toward locations where Google employees live and work. Speedometer collected the following measurements from 2011-10 to 2013-2 (17 months): 6.6M ping RTTs to `www.google.com` (each sample consists of 10 consecutive probes), 1.7M HTTP GETs to measure TCP throughput using a 224KB file hosted on a Google server, 0.4M UDP burst samples for measuring packet loss rate, 0.8M DNS resolutions of `google.com`, and 0.8M traceroute (without hop RTTs) from 144 carriers and 9 network technologies. The dataset includes ≈ 4-5 measurements per minute. Each measurement is annotated with device model, coarse-grained location information (k-anonymized latitude and longitude), timestamp, carrier, and network type.[3] All users consented to participate in the measurement study; the anonymization process is explained in the dataset's README file. Because of anonymization, the number of users who participated in data collection is unknown.

We augment the Speedometer dataset with 11 months of data collected by Mobiperf.

[1] http://www.mobiperf.com/

[2] This dataset is publicly available at https://storage.cloud.google.com/speedometer

[3] https://github.com/Mobiperf/Speedometer

Table 3.1: Number of Measurement and Carriers for the Network Technologies

	HSPA	HSDPA	UMTS	EDGE	GPRS	LTE	EVDO	eHRPD	1xRTT
# of Measurements	439K	2326K	563K	506K	58K	1460K	2183K	301K	68K
# of Carriers	50	111	96	85	48	7	8	2	3

Mobiperf conducts a superset of measurements in Speedometer, and notably adds signal strength information. The number of measurements collected by Mobiperf for each task ranges from 17K (HTTP GET) to 58K (ping RTT test) from 71 carriers. We use Mobiperf data to study the impact of signal strength on measurement results. Table 3.1 shows the number of measurements collected from the most frequently seen 9 network technologies (ordered by peak speed) for both GSM and CDMA technologies in the combined datasets.

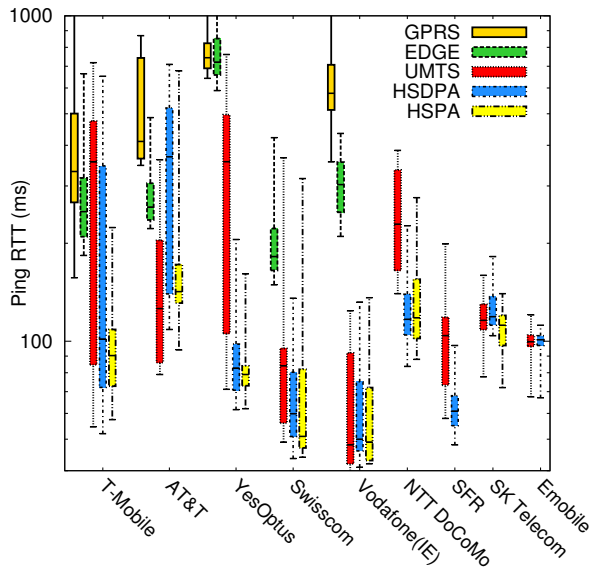
3.3 Data Analysis

3.3.1 Performance across carriers

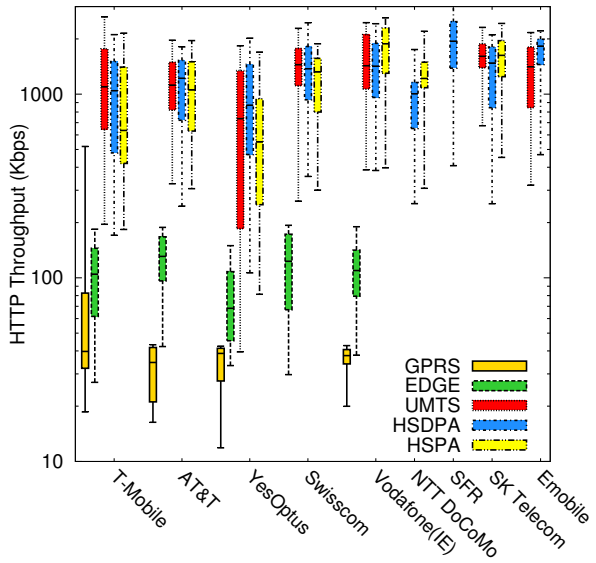
This section investigates the performance of five access technologies for each of several carriers. Our goal is to understand how observed performance matches with expectations across access technologies, and how variable this performance is across carriers. In Fig. 3.1, we plot percentile distributions (P5, P25, median, P75, and P95) of the latency and throughput of 9 carriers from Asia, America, Europe, and Australia. We select these carriers based on their geographic locations and relatively large data sample sizes. One of the key observations is that performance varies significantly across carriers and access technologies; further, the range of values is also relatively large.

For carriers that have high latency, we use traceroute data to investigate if the cause is inefficient routes to Google [190]. However, approximately half of the carriers such as SFR (French Carrier) and Swisscom have direct peering points with Google, making this unlikely to be the cause for high latency.

For carriers such as AT&T, T-Mobile US, and Airtel (India), we observe high variability in latency. In the following subsections, we investigate whether this is explained by regional



(a) Ping RTT



(b) HTTP GET throughput for downloading a 224KB file

Figure 3.1: Throughput and latency across access technology and carriers.

differences, time-of-day effects and/or other factors.

Surprisingly, we do not observe significant latency differences across access technologies for some carriers. For example, the latency of UMTS, HSDPA, and HSPA in Emobile (Ireland), SK Telecom (Korea), and Swisscom are almost equal. Users in these networks

may not see noticeable differences in performance for delay-sensitive applications when upgrading to newer technologies.

In Fig. 3.1b, we plot HTTP throughput for downloading a 224KB file from a Google domain. Compared to ping RTT, the difference between the throughput of carriers is relatively smaller, indicating that the high variability in ping RTTs is often amortized over the duration of a transfer.

Note that the throughput for UMTS, HSDPA, and HSPA are almost identical. This occurs because the flow size is not sufficiently large to saturate the link for high-capacity technologies. This indicates a need for better low-cost techniques to estimate available capacity in such networks [96]. However, the figure shows significant performance difference between GPRS/EDGE and other access technologies.

We observe that lower latency is generally correlated with higher HTTP GET throughput, but this depends on the carrier. We quantify this using the correlation coefficient between HTTP throughput and ping RTT for specific carrier and network type. The strongest correlation coefficient observed was for Verizon LTE users with -0.53 and lowest was -0.01 for T-Mobile HSDPA users, using one-hour buckets.

Having observed significant differences in performance within and between carriers, we now investigate some of the potential factors behind this variability.

3.3.2 Performance across different locations

We now investigate the impact of geography on network performance. We focus on four major US carriers in three US regions where our dataset is densest (New York, Seattle, and Bay Area). Each of these carriers exhibits different topologies (Internet egress, Google ingress and ASes between) in different regions, potentially leading to performance differences in each region.

Despite the variety in network topologies, we surprisingly find that for AT&T, T-Mobile, and Sprint, both of the latency and throughput were similar in these three loca-

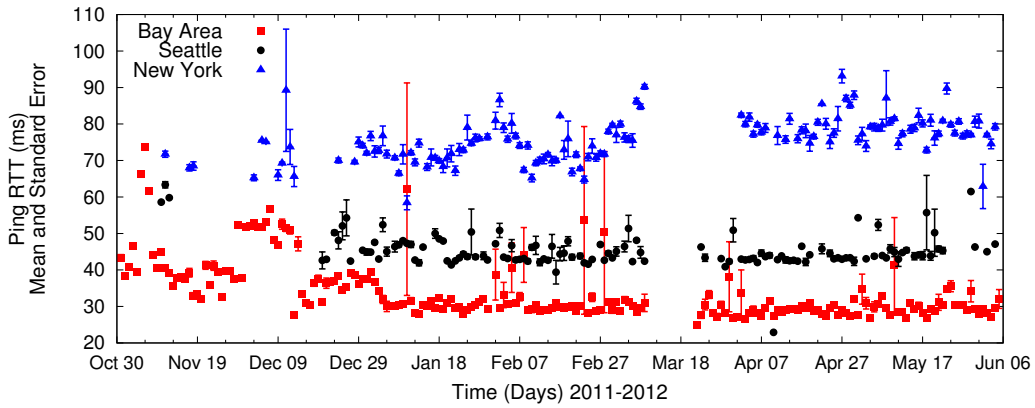


Figure 3.2: Verizon LTE Ping RTT in Different Locations

tions. However, for Verizon, we observe different LTE performance in New York, Seattle, and Bay Area. Fig. 3.2 plots these latencies over time, and clearly show that the RTT latency for the Bay Area is lower than New York and Seattle areas. HTTP throughput in these regions exhibit similar patterns.

We use DNS data in the Seattle area and observe that 97% of DNS requests for `google.com` resolve to an IP for a server in the Los Angeles area instead of Seattle, in part explaining the gap in latency between the two regions. For the NY area, our measurements did not provide enough geographic information to understand whether increased latency was due to path inefficiencies.

The key takeaway from this section is that geography alone doesn't explain the variance in performance observed in the previous section; however, for one carrier (Verizon), it explains some of it. Further, we observe that each region experiences changes in performance independently – the correlation of performance across regions for each carrier is negligibly small. Last, when correlating ping RTT and HTTP GET throughput within each region, we find higher correlations than carrier-wide correlations presented in the previous section. This further suggests that performance is affected by location.

3.3.3 Performance over time

We now analyze how performance depends on time – both in terms of time-of-day effects and the stability of measurement performance over time. These properties allow us to identify when to measure the network (*e.g.,* during known busy hours) and when *not* to measure (*e.g.,* at ten minute intervals), thus allowing us to efficiently allocate the limited measurement resources that users provide.

3.3.3.1 Time-of-day and long-term trends

Fig. 3.3 plots HTTP throughput for four major carriers in the US. As expected, throughput decreases (and variance tends to increase) during the busy hours for mobile usage (8AM to 7PM), likely due to higher load on the network. Interestingly, different carriers experience minimum throughput at different times. T-Mobile and AT&T reach their minimum throughput at 1PM and 5PM, respectively; Sprint experiences minimum performance at 9PM and Verizon, two troughs occur at 8AM and 9PM. Last, these carriers experience different relative variations in performance during busy hours: AT&T and Sprint throughput drops by approximately a third during busy hours while Verizon drops by 25%, and T-Mobile by 16%.

Next, we investigate the long-term performance trends over the duration of our study, allowing us to tell if new cellular technologies and infrastructure are keeping pace with increased mobile Internet usage. Specifically, we look at the change in throughput and latency of carriers through time over consecutive days for each network technology they support in different areas. We did not observe improvement; despite technology upgrades, performance is highly variable over time and there is no statistically significant change during the observation period.

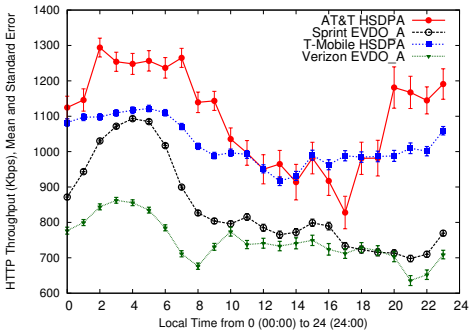


Figure 3.3: Time of day pattern of HTTP throughput

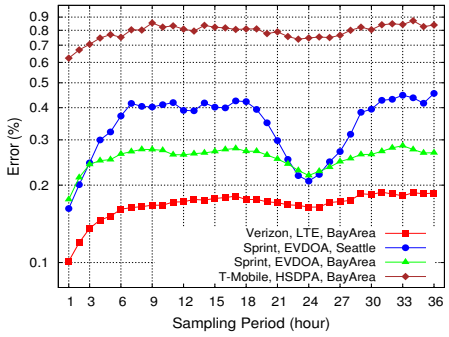


Figure 3.4: Weighted Moving Average Error (Median Ping RTT, $W = 2$)

3.3.3.2 Stability of performance

The predictability and stability of network performance are important not only for users, who are often frustrated more by variations in performance than the average value, but also for determining how and when to conduct measurements for future experiments. In this section, we compute stability using a weighted moving average and autocorrelation.

First, we group the data into 1-hour buckets (to obtain a sufficiently large sample size). Then for each bucket, we use either the median or 5th percentile latency. We compute the moving average error for different window sizes and sampling periods.

We compute the moving average error as follows: for a window size W, we predict the next data point on that series by computing moving average for the previous consecutive W points. For each W and sampling period (*e.g.,* every N hours for $N = 1, 2, 3, \ldots$), we compute the average over different offsets.

Fig. 3.4 plots the average error for all data points with windows size of 2 and different sampling periods for median ping RTT (results with larger window sizes of 3, 4, and 5 are similar). We observe that prediction accuracy varies significantly by carrier, with Verizon and Sprint in the Bay Area being relatively predictable, and T-Mobile and Sprint in Seattle being relatively unpredictable. Also, for all of these carriers, prediction accuracy is best when looking at the most recent data (one hour sampling period) and error tends to increase with longer durations, with the exception of 24hr (day) and 168hrs (week) sampling

periods, which are local minima. The results from autocorrelation are similar.

These predictability results indicate that despite the large overall variance in cellular network performance, there are regions and time scales over which performance is relatively predictable, depending on the carrier. Importantly, we can use this information to inform the design of measurement system that uses prediction to minimize probes that would provide redundant results. For instance, if we subsample every other value (*i.e.,* 50% sampling rate) in the Verizon LTE ping data in the Bay Area (which has the lowest error in the full sample), the distribution of latencies is nearly identical.

3.3.4 Performance Degradation: Root Causes

We now use our measurements to identify the reasons for persistent performance degradation observed in consecutive days. We focus on cases where the issue affects both ping RTT and HTTP throughput.

3.3.4.1 Inefficient paths

A reason for performance degradation is inefficient paths. Zarifis et al [188] provide a detailed taxonomy and analysis of path inflation in mobile networks; here we focus on their time evolution and constrain our analysis to only those cases where both latency and throughput were impacted.

For example, we observe an increase in ping RTT in T-Mobile's Bay Area HSDPA network from Nov 12, 2011 to Dec 10, 2011. Using DNS lookups, we find that clients previously sent to Mountain View were being sent to Seattle, with the additional delay explained by path inflation (Fig. 3.5a). After Dec 10, clients are again directed toward Mountain View.

We also observed a high-latency event for T-Mobile's Seattle HSDPA network in Seattle (Fig. 3.5b). Prior to the event, traceroutes indicate that traffic from T-Mobile ingresses into Level 3 in Seattle, then enters Google's network. After Feb 15, traffic from these sub-

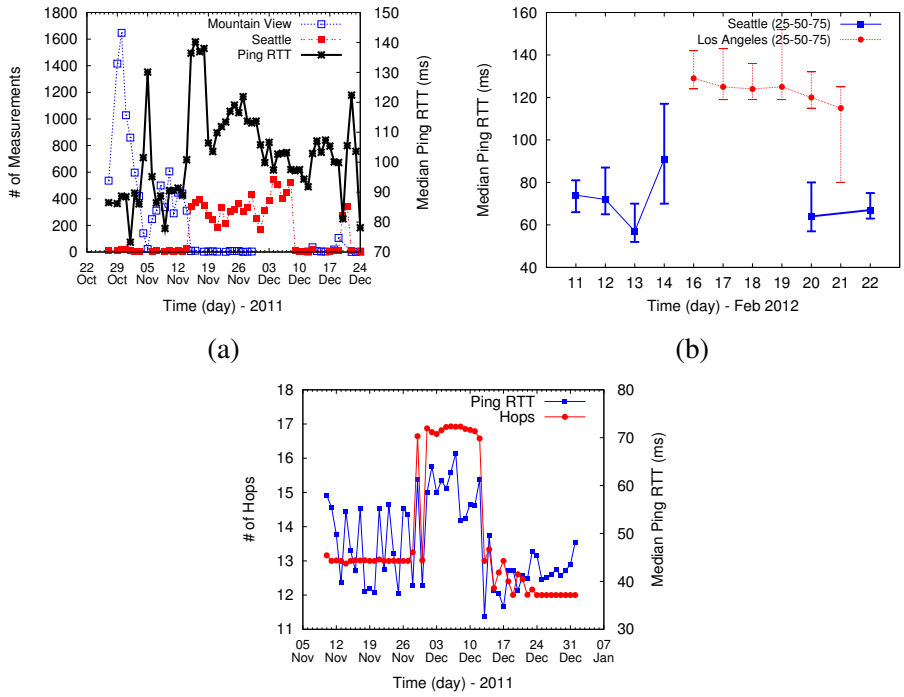
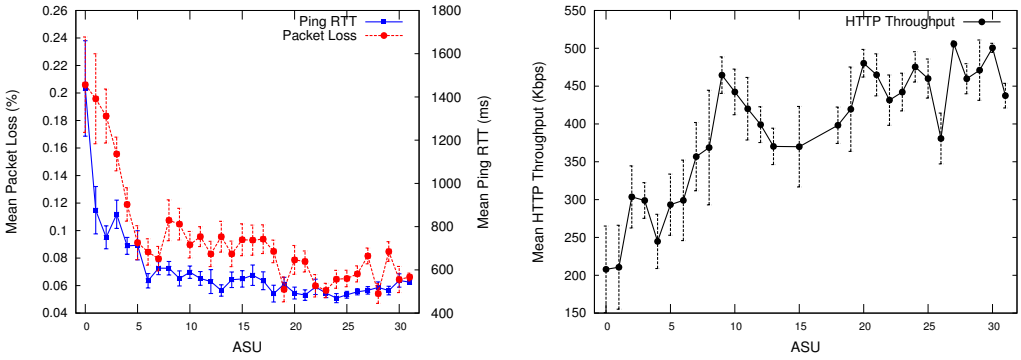


Figure 3.5: Performance Degradation in: (a)T-Mobile HSDPA network in Bay Area due to server selection flapping from Bay Area to Seattle (b)T-Mobile HSDPA network in Seattle due to change in ingress point of transit AS between T-Mobile and Google (c) Verizon LTE network in Bay Area.

scribers ingressed into Level 3 at a peering point in Los Angeles before entering Google's network. After Feb 20, routing returns back to its previous state (ingress and egress point in Seattle area) and the median RTT decreases to its previous value, strongly implying that the change in performance was due to the topology change.

In Fig. 3.5c, we observe that ping RTT and the number of traceroute hops increases for Verizon LTE users in the Bay Area. Previously, clients were sent to a Google frontend in the Bay Area; after the change clients are sent to the same Google ingress point, but then traffic is sent to a frontend in Seattle (leading to $\approx 30\%$ higher latency).

In this section we show that fixed-line inefficiencies can significantly impact the performance of LTE and HSDPA networks. For these newer technologies, since the RTT is lower, the impact of inefficient routes is even relatively higher (around 80% increase in the RTT of T-Mobile HSDPA in Seattle).



(a) Ping RTT and packet loss (b) HTTP throughput (file size is 100KB)

Figure 3.6: Impact of signal strength on latency, packet loss, and throughput.

3.3.4.2 Signal strength

It is well known that weak signal strength reduces channel efficiency for wireless communication; therefore, it is important to account for this when interpreting measurements. Using Mobiperf clients, we gather network measurements annotated with the signal strength, in Arbitrary Strength Units (ASUs),[4] reported during the probes and determine the impact of signal strength on performance.

Fig. 3.6 shows how three performance metrics vary with ASU values for AT&T HSDPA users in Seattle. The figures indicate high packet loss, latency and low throughput for ASU values between 0 and 8 (confirming the results in [160]); at larger ASU values that increase in signal strength has less impact on performance. These results indicate that accounting for signal strength is critically important for properly interpreting measurement results. For example, when measuring a carrier's capacity, it is important to do such tests in regions with high signal strength.

3.4 Conclusion

In this chapter, we took a first look at end-to-end performance as seen from mobile devices, using a dataset of scheduled network measurements spanning more than 100 carriers

[4] Android shows zero signal bar for the ASU values between 0 and 2 and full signal bars when ASU value is more than 12.

over 17 months. We find that there are significant performance differences across carriers, access technologies, geographic regions and over time; however, we emphasize that these variations themselves are not uniform, making network performance difficult to diagnose. Using supplemental measurements such as DNS lookups and traceroutes, we identified the reasons behind persistent performance problems. Further, we examined the stability of network performance, which can help inform efficient scheduling of future network measurements. Overall, we find that performance in cell networks is not improving on average, suggesting the need for more monitoring and diagnosis. As part of our future work, we are investigating how to automatically detect persistent performance problems in real time, gather additional network measurements to explain them and provide this information to carriers and end users automatically.

CHAPTER IV

Mobilyzer: An Open Platform for Controllable Mobile Network Measurements

4.1 Introduction

Given the tremendous growth in cellular data traffic, it is increasingly important to improve the availability, performance and reliability of the mobile Internet. To adequately address these challenges, we ideally would be able to collect network measurement data from any mobile device on any network at any time. With this information, users could evaluate the service they are paying for, carriers could study the factors impacting performance and detect problems causing degradation, and application developers could tune and improve their service. Many of these optimizations can be performed dynamically when data are available in real time.

Despite a need for performance improvement and policy transparency in this space [83, 137], researchers currently still struggle to measure, analyze and optimize mobile networks. Mobile Internet performance characterization is a challenging problem due to a wide variety of factors that interact and jointly affect user experience. For example, the performance that a user receives from an application can depend on the device's available hardware resources, radio's network state, the network access technology used, contention for the wireless medium, distance from cell towers, session initiation overhead, congestion at vari-

ous gateways and the overhead of executing code at the communicating endpoints. Further, this performance can change over time and as the user moves with the mobile device. Other challenges include resource constraints (data and power) on mobile platforms in addition to interference affecting measurements.

This problem has not gone unnoticed by researchers, operators and providers. A number of small testbeds and user studies have enabled progress in the face of these challenges [111, 46, 37, 65, 166], but with limited scope, duration, coverage and generality. We argue that previous work suffers from three key limitations that hamper their success. First, these individual solutions do not *scale*: each individual app or measurement platform is inherently limited to the population of participating users running a single piece of software. Second, each solution is *inconsistent* and *inflexible* in the set of network measurements it supports and the contextual information describing the experimental environment, making it difficult to ensure scientific rigor and to merge disparate datasets. Third, these solutions are *uncoordinated* in how they conduct network measurements: multiple apps can wastefully measure the same property independently or, worse, interfere with each other by running measurements at the same time from the same device.

Instead of proliferating apps that conduct independent network measurements in inconsistent ways, we argue that there should be a common measurement service that apps include in their code. Toward this goal, we designed and built *Mobilyzer*, a unified platform for conducting network measurements in the mobile environment. Our system is designed around three key principles:

1. **Measurement isolation:** Network measurements from mobile devices require tightly controlled access to the network interface to provide isolation from competing flows that impact measurement results.

2. **Global coordination:** Uncoordinated measurements from individual devices have limited scalability and effectiveness, so we provide a global view of available devices and their resources as a first-class service to support efficient and flexible measure-

ments without overloading any device or network.

3. **Incentives for researchers and developers:** Instead of focusing on a single "killer app" to obtain a large user base, we distribute the platform as a library to include in any new or existing app. Developers and researchers who write apps needing network measurement benefit from reduced operational cost and coding/debugging effort when using our library, and are given network measurement resources across *all Mobilyzer* devices in proportion to the number of users they convince to install their *Mobilyzer*-enabled apps.

This chapter answers key questions of how to (1) design a system that efficiently provides controllable and accurate network measurements in the mobile environment, and (2) leverage crowdsourcing to enable measurement studies previously infeasible in mobile networks.

Mobilyzer provides an API for issuing network measurements using a standard suite of tools, and manages measurement deployment and data collection from participating devices. Each device runs a measurement scheduler that receives measurement requests and gives experiments explicit control over the context in which a measurement runs. A cloud-based global manager dynamically deploys measurement experiments to devices according to available resources, device properties and prior measurement results.

Mobilyzer is currently deployed as a library that can be included in Android apps, and a cloud-based Google App Engine service for managing measurements across participating devices. *Mobilyzer* supports standard implementations of useful measurement primitives such as ping, traceroute, DNS lookups, HTTP GET, TCP throughput, and the like. These rich set of measurement primitives support experiments traditionally popular in fixed-line networks, such as mapping Internet paths (via traceroute), measuring and comparing performance for different destinations, and understanding broadband availability. We also support advanced application-layer and cellular-specific measurements, such as inferring RRC timers, measuring Video QoE, and breaking down Web page load time into its con-

stituent dominant components.

Our design supports a constrained form of programmable measurements: an experiment may consist of sequential and/or concurrent measurements, where the execution of subsequent measurements can depend on the results of prior results and contextual information (*e.g.,* signal strength or location). We demonstrate the flexibility of these *compound measurements* using several experiments as case studies. For instance, we use this feature to trigger diagnosis measurements when anomalies in network performance are detected, or when we predict that a handover might occur.

We evaluate our deployed system in terms of our design goals. We demonstrate that it effectively provides measurement isolation, and failure to do so can lead to unpredictable and large measurement errors. Further, we show that *Mobilyzer* manages measurement scheduling efficiently, adapts to available resources in the system, is easy to use, and reduces development effort for measurement tasks.

We further evaluate our system in terms of new measurement studies that it enables, using *Mobilyzer*'s support for coordinated measurements among large numbers of vantage points to evaluate the performance of Internet-scale systems. We use *Mobilyzer* to identify and diagnose cases of CDN inefficiency, characterize and decompose page load time delays for Web browsing, and evaluate alternative video bitrate adaptation schemes in the mobile environment. For example, we find that (1) poor CDN replica selection adds 100 ms or more latency in 10% of our measurements, (2) limited mobile CPU power is a critical bottleneck in Web page load times (typically between 40-50% of load times) and doubling CPU speed can reduce load times by half, and (3) buffer-based adaptive video streaming improves the average delivered bitrate by 50% compared to commonly used capacity-based adaptation for low bandwidth clients.

Table 4.1: Measurement types supported by *Mobilyzer*.

Measurement Type	Supported Measurements	Usage
Basic	DNS lookups, ping, traceroute, HTTP GET, TCP throughput, and UDP burst	Supports experiments traditionally popular in fixed-line networks, such as mapping Internet paths, measuring and comparing performance for different destinations and carriers, and understanding broadband availability
Composed	Sequential and parallel	Combines multiple measurements programmatically to support new measurements, such as diagnosis measurements
Complex	RRC timer inference, Video QoE, and Page load time measurements	Supports application-layer and cellular-specific active measurements

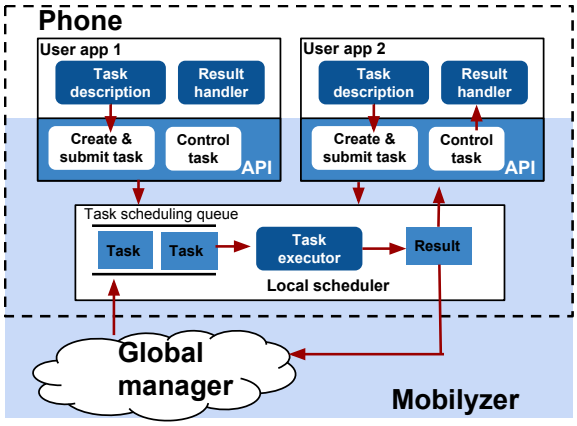


Figure 4.1: *Mobilyzer* architecture (shaded region). Apps (top) include the *Mobilyzer* library, which provides an API to issue network measurements, and a scheduler service (middle) that provides measurement management and isolation. The scheduler communicates with a global manager (bottom) to fetch measurement requests and report measurement results.

4.2 Goals

The goal of *Mobilyzer* is to provide a scalable platform for conducting meaningful network measurements in the mobile environment. This is the first open-source platform[1] to support mobile measurements in the form of a library. It is designed to achieve:

4.2.1 Standard, Easy-to-use Measurements

There is a tension between arbitrary flexibility and standardization in a platform for network measurements. While allowing experimenters to execute arbitrary code within a sandbox provides flexibility, a lack of standards means that these experiments are difficult to incorporate into existing and future datasets. In *Mobilyzer*, we opt for standard measure-

[1] *Mobilyzer* source code is publicly available at http://mobilyzer-project.mobi/

ments, facilitating comparative and longitudinal studies atop our platform (see §4.3.2).

4.2.2 Measurement Isolation

The mobile environment poses unique challenges for controlling how network measurements are isolated from each other, and from network traffic generated by other apps and services. *Mobilyzer* accounts for these properties and gives experimenters explicit control over device state and concurrent network activity during measurement. *Mobilyzer* achieves the goal of measurement isolation via a local measurement scheduler (see §4.3.3).

4.2.3 Global Coordination

Data quota and power are scarce resources for mobile networks, requiring that researchers use more principled approaches than "shotgun" measurement. With a global view of available network resources and the ability to coordinate measurements from multiple devices, we can coalesce redundant measurements and allow researchers to specify targeted experiments that use device resources efficiently (see §4.3.4). For instance, coordination among the *Mobilyzer*'s clients provides opportunities to (1) evaluate the performance of Internet-scale services and (2) schedule measurements in an interactive fashion, where a set of measurements scheduled on one device may depend on the results *Mobilyzer* receives from other devices.

4.2.4 Incentives for Adoption

As a crowdsourcing approach, *Mobilyzer* requires a crowd of mobile users to adopt our platform. We argue that no single app will provide the necessary crowd. Instead, we opt for a "bring your own app" deployment model, where researchers/developers develop *Mobilyzer*-enabled apps with user incentives.

We propose the following incentives for researchers and developers to use our library.

- *Incentive for researchers.* The incentive for researchers to develop *Mobilyzer*-enabled apps is that they can conduct measurements on *any* of the devices in the system (including those not running their app), with measurement quota proportional to the number of devices their app(s) bring to the system. This is analogous to the PlanetLab and RIPE Atlas models, except using a software (instead of hardware) deployment.

- *Incentive for developers.* One incentive for adopting *Mobilyzer* is reduced operational costs. As an example, an app that provides "speed tests" of peak transfer rates can incur significant bandwidth charges for the servers hosting transferred content. Our system uses resources donated by M-Lab, and this was the main reason that MySpeedTest adopted our tool.

  A second incentive is reduced coding/debugging effort. Supporting any app that requires network measurement (and possibly access to measurements from multiple devices) entails writing and maintaining software for measurement and data storage/retrieval. Our experience is that correct implementations of such functionality can take man-months to man-years of work. We provide this functionality out-of-the-box with *Mobilyzer*.

  We also consider altruism and relationships with developers as incentives. For example, the Insight [139] project successfully convinced developers of a popular app to instrument and share data with researchers. Similarly, altruism has attracted users to support platforms such as RIPE Atlas and DIMES.

Comparison with ad libraries. Our library-based deployment model is inspired in part from the successful adoption of advertising (ad) libraries in mobile apps. We believe that ad libraries are successful in large part because they have direct incentives for developers (payment for click-throughs) and a limited impact on apps. Our direct incentive for developers is that they "earn" measurement resources proportional to the size of the *Mobilyzer*-enabled app deployment. This is analogous to the RIPE Atlas credit model, which has helped grow a hardware-based platform to more than 7,000 active hosts. Like ad libraries,

which periodically pull advertisements for display, *Mobilyzer* primarily uses a pull-based model to fetch measurements.

However, ad libraries and *Mobilyzer* differ from the perspective of privacy and resource usage. Ad libraries can be used to track arbitrary information about users, often without any explicit privacy policy or user awareness. *Mobilyzer* is explicit about the data it collects, was designed with privacy concerns in mind, and requires explicit user consent. Further, it is unclear if or how ad libraries limit the resources consumed by their service. *Mobilyzer* provides controllable, limited impact on apps by enforcing hard limits on resource consumption (in terms of energy and data quota) and avoiding interference with network traffic from other apps. While *Mobilyzer* provides direct incentives for developers in terms of measurement resources, unlike ad libraries it will not directly subsidize app development costs.

4.2.5 Nongoals

First, we do not provide a "PlanetLab for mobile"; i.e., we do not provide a platform for deploying arbitrary distributed system code on mobile devices. Instead, we focus on the more constrained problem of providing a platform for principled network measurements in lieu of one-off, nonstandard measurement studies. Second, we do not provide a service for arbitrary data-collection from mobile devices; rather, we focus on active and passive network measurements annotated with anonymized contextual information. Collecting arbitrary data from users is a potential privacy risk that we avoid in *Mobilyzer*. Third, we do not propose any specific incentives for device owners to adopt *Mobilyzer*-enabled apps. Rather, we urge experimenters to either develop apps with user incentives in mind, or convince maintainers of existing popular app codebases to include our library. Currently, there are hundreds of apps [20] that simply do network measurements (e.g., speed tests) or network diagnosis (signal strength/coverage maps) and have large user bases. We believe this is strong evidence that users will install a *Mobilyzer*-enabled tool as long as it is useful.

Table 4.2: *Mobilyzer* key API functions to support issuing and controlling measurement tasks.

API Function	Description
`MeasurementTask` **`createTask`**`(TaskType type, Date startTime, Date endTime, double intervalSec, long count, long priority, Map<String,String> params)`	Create a task with the specified input parameters including task execution frequency task priority, etc.
`String` **`submitTask`**`(MeasurementTask task)`	Submit the task to the scheduler and return `taskId`.
`void` **`cancelTask`**`(String taskId)`	Cancel the submitted task, only allowed by the application that created this task.
`void` **`setBatteryThreshold`**`(int threshold)`	Set the battery threshold for check-in and running the global manager scheduled tasks.
`void` **`setCheckinInterval`**`(long interval)`	Set how frequently the scheduler checks in.
`void` **`setDataUsage`**`(DataUsageProf profile)`	Set a limit for *Mobilyzer*'s cellular data usage.

We also believe that *Mobilyzer*'s limited data collection is not a strong impediment to user adoption, as ad libraries are known to collect more intrusive information with no known effect on adoption.

4.3 Design and Implementation

4.3.1 Overview

Figure 4.1 provides a high-level view of the Mobilyzer platform. It consists of an app library, a local measurement scheduler, and a cloud-based global manager.

Measurement Library. *Mobilyzer* is designed to be easy to use and integrate into existing apps, while providing incentives for doing so. It is deployed to apps as a library with a well-defined API for issuing/controlling measurements and gathering results locally. This facilitates development of new apps that use network measurements. For example, an app that includes our library can issue a measurement and retrieve results with under 10 lines

of code. Existing apps can incorporate our library to take advantage of our validated measurement tools and server-side infrastructure.

Local measurement scheduler. *Mobilyzer* achieves the goal of measurement isolation via the local measurement scheduler. The scheduler listens for measurement requests and ensures that any submitted measurements are conducted in the intended environment (*e.g.,* radio state, state of other apps using the network, and contextual information such as location, signal strength and carrier). Regardless of how many apps on a device use it, *Mobilyzer* ensures that there is exactly one scheduler running. If apps with multiple versions of the scheduler are present, the latest version is guaranteed to be used. The scheduler enforces user-specified limits on resource (data/power) consumption.

Global manager. The *Mobilyzer* global manager provides coordination for the following services: dynamic measurement deployment, resource cap enforcement, and data collection. This support is quite unique to our platform, as global centralized coordination improves the design of mobile experiments despite limited resources. Experimenters use the global manager to deploy measurement experiments to devices. For example, an experiment may contain a number of network measurements that should be conditionally deployed and executed according to device properties. The global manager deploys these measurements to devices based on device information reported periodically, and ensures that measurements are not deployed to devices that have exceeded user-specified resource caps. The manager maintains a datastore of anonymized data collected from all devices and makes this available for interactive measurement experiments and for offline analysis. The data collection and scheduling support at the global manager enable dynamically triggered measurements based on observed network behavior, achieving a feedback loop of measurement, analysis, and updated measurements.

4.3.2 Measurement Library and API

Design. A key design principle for *Mobilyzer* is that the platform should remove barriers to widespread adoption so that it facilitates a large-scale deployment. We chose to implement *Mobilyzer* as a network measurement library that app developers include in their code. This code provides an API for issuing measurements using a standard suite of tools (Table 4.2), and a device-wide measurement scheduler.

The measurement model supported by *Mobilyzer* represents a trade-off between complete flexibility to run arbitrary code and complete safety in that all resources consumed by a measurement can be predicted in advance. Specifically, *Mobilyzer* supports a small set of commonly used *measurement primitives*, e.g., ping, traceroute, DNS lookup, HTTP get and throughput tests. These measurements can be performed independently in isolation, or they can be chained (do a DNS lookup for google.com, then ping the IP address) or executed in parallel (e.g., ping google.com during a TCP throughput test) in arbitrary ways.

Using this model, the amount of data and power consumed by each task (simple or complex) is predictable with few exceptions (e.g., downloading an arbitrary Web page) that can be mitigated via scheduler-enforced constraints on the data and power consumption of a task. Like Dasu, this approach avoids concerns from running arbitrary code; however, unlike Dasu this allows us to strictly control resource consumption (which is not a primary goal of Dasu).

Implementation. The *Mobilyzer* measurement library is implemented as Android code that is added by reference to an existing app's source code. *Mobilyzer* supports three types of measurements: *Basic*, *Composed*, and *Complex Measurements (Table 4.1).*

Basic Measurements Supported. Mobilyzer supports both passive, contextual measurements and active network measurements. Currently, the active measurements supported are DNS lookups, ping, traceroute, HTTP GET, and a TCP throughput and UDP burst test. For each measurement task, users can specify general task parameters (*e.g.,* the time at which to run) and task specific parameters (*e.g.,* TTL value for ping measurement). Fur-

ther, each task can specify a *pre-condition* which must evaluate to true before a task can be executed (*e.g.,* location is Boston, signal strength is greater than 50, network type is cellular).

Mobilyzer also collects passive measurements on both network performance and device state, and associates a set of passive measurements with every active test run. It collects the signal strength (RSSI values) and the device's battery level, battery charging state, coarse-grained location, the network technology, the total number of bytes and packets sent and received by the device, as well as static context information such as the carrier, OS, and support for IPv6. Our set of measurements and monitored device properties do not require special privileges; however, we can support measurements that require rooted phones (if available).

Measurement composition. Mobilyzer supports combining multiple measurements into a single experiment, where sets of measurement tasks can run sequentially and/or in parallel. Further, the result of each sequential task can be used as a pre-condition for executing a subsequent task. This feature improves experiment flexibility—we use this feature to trigger diagnosis measurements in response to detecting anomalies in network performance. Figure 4.2 shows how this feature can help in building a simple diagnosis task, where the client will run traceroute, if latency and signal strength is above a threshold.

Supporting complex measurements. Mobilyzer is designed to provide extensible support for application-layer and cellular-specific measurements. These include inferring RRC timers, measuring Video QoE, and breaking down Web page load time into its constituent dominant components. Further, our system can incorporate new complex measurements as they become available. Each new task must have predictable power consumption, data usage, and duration, so that *Mobilyzer* can ensure that the measurement task will not exceed a device's battery and data usage limits.

We now describe how long-running, complex tasks motivate the need for additional scheduler features to ensure measurements complete successfully. Mobile devices change

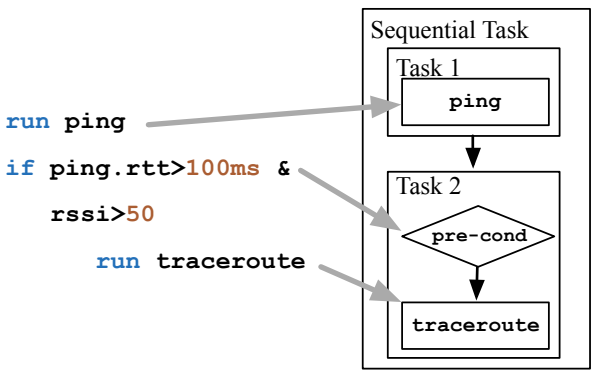
between different power states, called RRC states, in response to network traffic [149, 100]. The device's RRC state is not exposed to the OS due to a lack of an open API, so identifying how long devices stay in each power state entails sending a large number of individual packets with long gaps between them, and inferring power states based on observed packet delivery delays.

This type of measurement poses two key challenges. First, it is a long-running, low-rate measurement that should run only when connected to a cellular network. With no other tasks running on the device, the test can easily take half an hour. Given our pre-emptive priority scheduler (described in the next section), the RRC inference task will either block other tasks for long durations or will be constantly interrupted by higher priority tasks. The latter situation can lead to a large volume of wasted measurement bandwidth (as interrupted measurements must be discarded and restarted), and with sufficient interruptions the RRC task may not have a chance to complete.

Second, this task requires complete isolation from other network activity. Any concurrent traffic will alter the results of this test by changing the radio power state. To detect whether the network isolation property holds, *Mobilyzer* collects information about device-wide background traffic through the *proc* file system and discards results taken when there is background traffic.

These challenges motivate the following approach for the RRC state task. First, it runs with low priority so it does not block other measurements for unreasonable periods. Second, because it may be frequently pre-empted or unable to run due to the device using WiFi, we support suspending and resuming this task. Third, we upload partial measurement results when the task fails to complete, so the data collected is not wasted. We used this measurement to understand the impact of RRC states on latency and packet loss for various network protocols and applications, and found the presence of unexpected application-layer delays [156].

More generally, we include these features for any task that logically supports suspend,




```
params = new HashMap<String, String>();
params.put("target", "www.google.com");
MeasurementTask pingTask =
    api.createTask(PingTask.TYPE,
    params);
MeasurementTask tracerouteTask =
    api.createTask(TracerouteTask.TYPE,
    params);
tracerouteTask.setPreCondition(AndCriteria(
  Criteria(Criteria.RESULT, 0
     ,"mean_rtt_ms",">",100),
  //0 is index of prior task
  Criteria(Criteria.RSSI,">",50)));
MeasurementTask seqTask =
    api.createTask(SequentialTask.TYPE,
    Config.DEFAULT_MEASUREMENT_COUNT,
    Config.DEFAULT_MEASUREMENT_INTERVAL_SEC,
    API.USER_PRIORITY);
seqTask.setSubTasks([pingTask,tracerouteTask])
api.submitTask(seqTask);
```

Figure 4.2: Using sequential tasks to build a simple diagnosis task.

resume and partial measurements. For example, traceroute includes these features but atomic measurements such as ping do not.

4.3.3 Local Measurement Scheduler

Design. The local measurement scheduler is a service running on a device that manages the execution of measurement tasks. It can be implemented in the operating system or run as a user-level background service. The scheduler provides a measurement execution environment that can enforce network isolation from other measurements and apps running on the device. It also enforces resource-usage, priority, and fairness constraints.

Task priorities. The scheduler supports measurement task priorities with pre-emption. It executes tasks with the highest priority first and will pre-empt ongoing measurements if they are of lower priority. Certain measurement tasks cannot produce correct results if pre-empted (*e.g.,* a ping task pre-empted between sending an echo request and receiving a response), so pre-emption will lead to wasted measurement. To balance the trade-off between this waste and scheduler responsiveness to high-priority tasks, the scheduler waits

for a short time (*e.g.,* one second) before pre-empting a measurement.

This is sufficient time for a ping measurement to complete, but not necessarily for other tasks (*e.g.,* traceroute). To minimize wasted measurement resources for pre-empted tasks, we define a `pause()` method that signals to measurement tasks that they should save their state due to an imminent pre-emption. For traceroute, this means that the traceroute measurement can save its current progress and continue to measure a path once it is rescheduled.

Not all measurements can (or should) be pre-emptible. In addition to the traceroute task mentioned above, we have also implemented a pre-emptible long-running task for inferring RRC state timers on mobile devices [149, 100]. Measurements like DNS lookup and ping are not pre-emptible (i.e., they are killed at pre-emption time) because they are short-lived tasks with minimal (or no) state to cache. In general, network measurement tasks (e.g., throughput measurement) to characterize time-varying properties of the network cannot benefit from saved state when pre-empted; this is in contrast to measurements of more stable properties (e.g., RRC state machine and NAT policies).

Traffic Interference. To avoid inference from (and with) traffic from other apps, *Mobilyzer* monitors traffic generated by other apps (using the TrafficStats API in Android) and pauses/kills measurement tasks that are subject to interference. For example, the TCP throughput task, which attempts to saturate bandwidth to measure throughput, will be stopped if scheduler detects any concurrent network traffic; failing to do so may adversely affect other applications' performance and lead to inconclusive measurement results.

Resource constraints and fairness. The scheduler accounts for how much data and power is consumed by network measurements and ensures that they stay within user-specified limits. Users specify hard limits on how much data *Mobilyzer* consume, and at what battery level the library should stop conducting measurements.

We enforce limits as follows.

- The scheduler does not execute a task if its data or energy consumption estimates ex-

ceed available resources. Such tasks are suspended until the constrained resource(s) is replenished. For energy, this happens when a device is recharged; for data plans, this is typically done according to the billing cycle.

- The scheduler monitors data and power consumption for ongoing measurements, and terminates a task if it consumes more data than expected, and/or exceeds a device's data/power quota. Because measurement tasks cannot execute arbitrary code, we can always predict in advance the maximum resources they may consume, and use that as a conservative estimate for scheduling purposes.
- To ensure liveness and fair access to resources, the scheduler enforces an upper bound on task duration by terminating execution of a measurement task exceeding that limit. This limit must be specified in the task description.

All measurement tasks generated by apps running on a device are assigned the same high priority value. To ensure fairness among multiple apps on the same device competing for resources, we ensure that each app receives an equal share of the constrained resource.

Implementation. The scheduler is currently implemented using an Android service, which guarantees that at most one scheduler is present regardless of the number of *Mobilyzer*-enabled apps running on a device. The library communicates with the scheduler service via IPC calls.

Forward compatibility. Our library-based deployment model means that different *Mobilyzer*-enabled apps running on the same device may use different versions of the measurement library and scheduler. We want to ensure that all apps on the device bind to the *newest* version of the scheduler service as soon as it is installed by an app. A key challenge for providing this functionality is that by default on Android, apps will bind only to the scheduler of the app that is first installed, which is unlikely to be the latest version.

We address this as follows. First, we exploit the Android priority tag in the *bind* Intent when declaring the scheduler in the manifest file. The scheduler with the highest priority is bound if there is no other bound scheduler, so we increment the priority value with each

new *Mobilyzer* version. However, if an older scheduler has already been bound, apps bound to an older version do not switch until the device is rebooted. To address this, we deliver the scheduler version information via the *start* Intent. When the scheduler receives a *start* Intent, it compares its version with the one in the intent and terminates if it sees a newer scheduler version.

4.3.4 Global Manager

Design. The *Mobilyzer* manager maintains a global view of measurement resources, efficiently dispatches experiment tasks to available devices and coordinates measurements from multiple devices to meet various experiment dependencies. Atop the manager is a Web interface that allows experimenters to specify measurement experiments that are deployed to devices.

Specifically, researchers can submit a measurement schedule, along with information about the properties of devices that should participate in the experiment (e.g., location, device type, mobile provider). Devices periodically check in with the manager and receive a list of experiments to run. When complete, devices report the results of the measurements to the manager.

The manager ensures: (i) no experiment uses more than its fair share of available measurement capacity across *Mobilyzer* devices; (ii) experiments are scheduled in a way that maximizes the likelihood that the targeted devices will complete their measurements during the period of interest; and (iii) it enforces limits that prevent harm to the network or hosts (*e.g.,* via a DDoS).

Fairness under contention. In a successful *Mobilyzer* deployment, there are likely to be cases where measurement requests exceed available measurement resources. In periods of contention, we must dispatch measurement tasks according to some fairness metric.

We assign measurement tasks to devices such that the fraction of assigned tasks for an experimenter is proportional to the number of measurement-enabled devices that the exper-

imenter has contributed to *Mobilyzer*. When measurement demand from an experimenter is lower than their fair share, we backfill available resources from remaining pending tasks to ensure that our system is work conserving. Similar to previous work [54, 127], we can use an auction-based approach as a building block to provide such fairness.

Availability prediction. Available devices in *Mobilyzer* are subject to high churn and mobility, meaning that simply deploying an experiment to some random fraction of devices may lead to a low rate of successful measurements. To improve this, the global manager incorporates knowledge of experiment dependencies, prediction for resource availability and accounting for failures due to issues such as disconnections and loss of power. For example, we use prediction to prevent wasted measurement resources by preventing experiments from being dispatched if they are likely to fail (*e.g.,* due to unavailable measurement quota or due to measurement preconditions not being met on the device).

Other features. Mobilyzer supports a variety of other features, described in detail in the technical report [133]. Briefly, we support live measurement experiments, which allows researchers to specify measurement experiments that are driven by results from ongoing measurements; i.e., experiments that cannot be specified a priori (demonstrated in §4.4.4.1). The global manager can prevent harm to the network and to other hosts by accounting for all measurement activity in the system and ensure that no host or network is overloaded by measurement traffic. Last, users can access their data by logging into our dashboard [24], and collected data is anonymized and publicly accessible online [25].

Implementation. The *Mobilyzer* manager is currently implemented using a Google App Engine (GAE) instance, providing a highly scalable platform for managing thousands or millions of devices. The manager currently uses a pull model for experiment deployment, where devices check in with the manager periodically to retrieve updated experiments and to update their contextual information (coarse location, remaining battery, access technology). We also support Google Cloud Messaging (GCM) services to provide a push-based model for experiments with tighter timing constraints, which is subject to the same resource

constraints as pull-based measurements.

Experimenters currently can specify simple measurement schedules using an interface that permits control over measurement parameters, experiment durations and periodicity. We have also developed several dynamic measurement experiments as described in Section 4.4.4, where the measurements issued to a device depend on the results of prior measurements (from potentially many other devices).

The manager deploys measurements based on contextual information gathered from devices. For example, the manager will reduce the measurement rate for periodic tasks that would otherwise exceed device quota. Similarly, measurement tasks for devices in a specific geographic region are not deployed to devices outside that region.

4.3.5 Security

Security is paramount in any large, controllable distributed system. Our design addresses the following threat model: an attacker who subverts the system by taking over our control channel, hacking our global manager, participating in the system to launch as DDoS, or control/drain resources on devices. We address these attacks as follows:

- **Subverting the control channel:** We use Google App Engine (GAE) to control our devices, which relies on HTTPS to communicate with the trusted GAE hosts. Subverting the control channel requires subverting the PKI for Google certificates, which is difficult.
- **Subverting the GAE controller:** We use Google account authentication to control access to our global manager and limit access only to authorized users. This does not address account compromise, which we cannot rule out. However, we monitor system usage, which should allow us to react quickly should this occur.
- **Insider attack:** A malicious developer could try to use our system to launch "measurements" that actually consist of a DDoS attack or resource drain. To prevent the former, we account for all measurement requests scheduled either by check-in or push-based

mechanism and conducted by all participating devices, and place limits on how many measurements can target a given domain or network. To prevent the latter, we use the hard limits on resource constraints mentioned above.

- **App compromise:** An attacker may wish to take control of apps/devices by executing code that compromises the app/device OS. We support executing only measurements we validate, limiting the attack surface for compromise. We further rely on the Android/Java sandboxes to prevent OS compromise.

4.4 Evaluation

We now evaluate *Mobilyzer* in terms of deployment experience, performance, and applications.

4.4.1 Deployment Experience

One key advantage of *Mobilyzer* is that apps requiring network measurements are easier to write and maintain. Our platform provides validated measurement code, prevents interference from other *Mobilyzer*-enabled apps, collects and stores measurement data and utilizes existing infrastructure (via M-Lab) for bandwidth testing. We believe that by separating measurement management and data collection code from app development, *Mobilyzer* helps researchers focus on interesting measurement experiments and compelling incentives for user adoption.

Table 4.3 lists the lines of code (LoC) used for network measurements in three popular measurement apps. In Mobiperf, 80% of the code was for measurement. Using a library-based model, we simplify the app codebase, making it easier to focus on UI and other elements to encourage user adoption. With *Mobilyzer* one can issue a measurement task and retrieve the results with fewer than 10 LoC.

Mobilyzer has been adopted by the developers of MySpeedTest, a throughput-testing

Measurement App	Measurement LoC
FCC SpeedTest [83]	12550
MySpeedTest [123]	9545
Mobiperf	8976

Table 4.3: Lines of code (LoC) for open-source measurement apps. By integrating *Mobilyzer* into existing apps, developers can save thousands of lines of code.

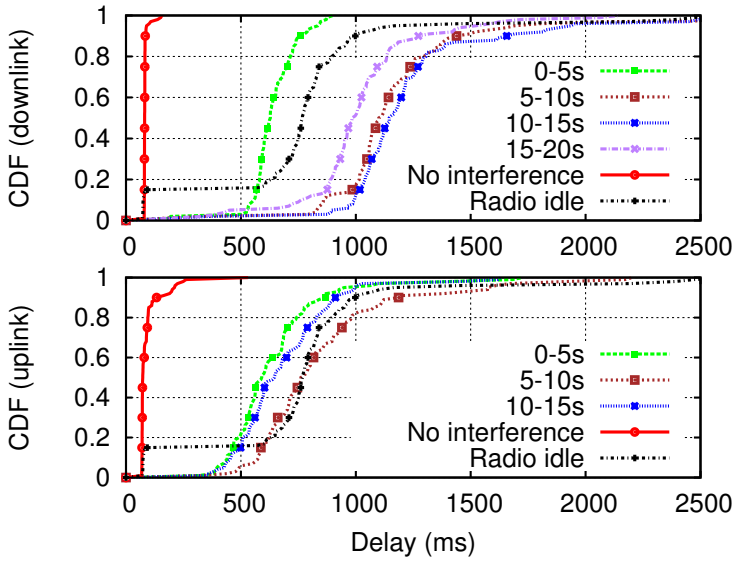


Figure 4.3: Impact of throughput measurements (grouped into time periods) and radio state on measured RTT. Results vary with radio state, and upstream or downstream cross-traffic with varying throughput. *Mobilyzer* provides strict control over these kinds of dependencies.

app. The experience from the main developer was overall quite positive, and identified cases where instructions for use were unclear. While it took several e-mail exchanges to clarify these issues, the developer reported that writing the code to port MySpeedTest to *Mobilyzer* took "about an hour or less" and they have used our library since May, 2014. We are currently in discussions with other researchers about integrating *Mobilyzer* into their apps.

4.4.2 Measurement Isolation

An important feature of *Mobilyzer* is that it schedules measurements to provide applications with control over if and when measurements are run in isolation. We now use

this feature to demonstrate the value that isolation brings. In particular, we use a compound measurement task that consists of a TCP throughput test run in parallel with a ping measurement. We vary the start time for the ping measurement such that it occurs before, during, and after the throughput test and plot the results. We also control whether the cellular radio is in a low-power state before measurement. Each experiment is repeated 40 times; we plot a CDF of ping latencies for each configuration. We omit throughput results because they are unaffected by ping measurement cross traffic.

Figure 4.3 shows that ping measurements are significantly affected by cross traffic, and the difference in latency compared to the case with no interference can be hundreds of milliseconds or seconds. The additional delay, presumably from queuing behind the TCP flow from the throughput test, also varies depending on how much cross traffic occurs. As a result, interference from cross traffic essentially renders the latency measurement meaningless. Note that the downlink interference is more severe likely due to higher throughput (2.5 Mbps down vs. 0.35 Mbps uplink), which leads to more queuing inside the network and on the device.

The figures also show differences in measured latencies when the radio is active compared to when it is idle before measurement. Similar to the above scenario, not accounting for this effect can cause misleading or incorrect conclusions. With *Mobilyzer*, experimenters can tightly control the impact of these sources of noise in measurement experiments.

4.4.3 Microbenchmarks

This section presents results from controlled experiments evaluating *Mobilyzer* overhead in terms of measurement-scheduling delay, power usage, and data consumption.

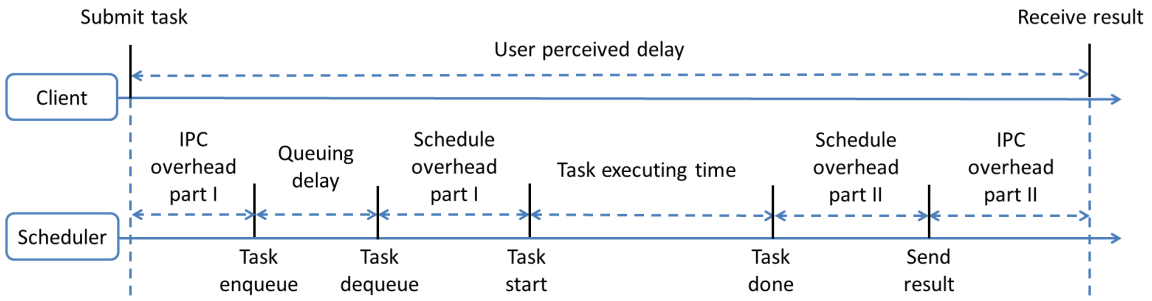


Figure 4.4: Scheduling overhead for a *Mobilyzer* measurement task.

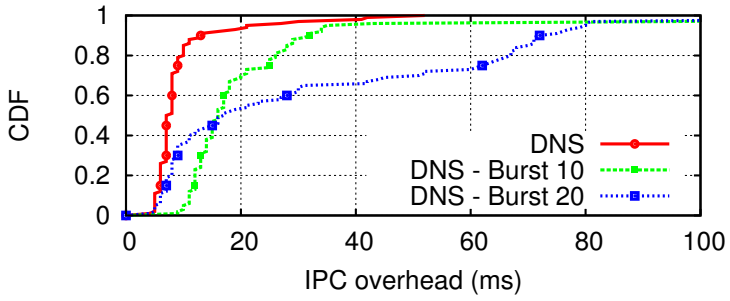


Figure 4.5: IPC overhead for DNS burst sizes.

4.4.3.1 Scheduling Delays

The scheduling delay introduced by *Mobilyzer* consists of the delay from using IPC (interprocess communication) between an app and the scheduler, and the delay introduced by the scheduler. As we show, these delays are reasonably low for all measurements and under significant load.

The IPC delay (Fig. 4.4) is the sum of (i) the delay between when the client submits a task and when the server receives it (IPC part I in the figure), and (ii) the delay between when the scheduler sends the result to the client, and the client receives it (IPC part II).

We use an HTC One (Android 4.1.1, using LTE) to run measurement tasks for characterizing the IPC overhead. Each task is run 100 times. Fig. 4.6 presents a CDF of the delay; for all tasks, the maximum delay is below 100ms, while most delays are within 20ms. These values match the performance analysis for IPC latency using Intents [95]. Note that this delay affects the time until a measurement runs but does not interfere with

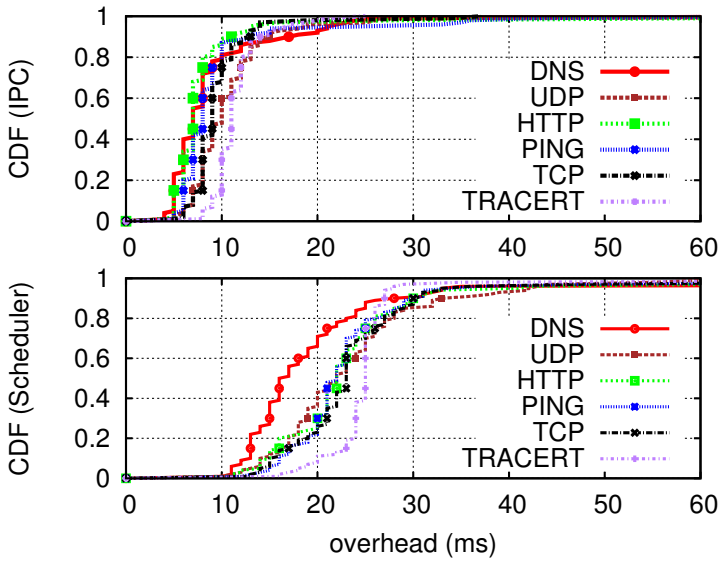


Figure 4.6: IPC and schedule overhead, 1 app.

measurement execution.

Note that the IPC overhead increases when many tasks are submitted back to back, due to the way Android implements Intent-based IPC. Specifically, there are two IPCs for each Intent-based IPC – one from sender to the Intent manager, and then one from the manager to the receiver [95].

To test the impact of this, we submit bursts of 1, 10 and 20 tasks with 15 ms between each burst. If several IPCs are sent to the Intent manager at once, there may be a delay in redirecting them to the receiving app. Fig. 4.5 shows the IPC overhead under high load for the DNS task (other tasks exhibit a similar pattern). Although larger burst sizes lead to longer delays, with a burst of 10 the delay is mostly within 50 ms. For a burst of 20, over 90% are below 100 ms, which we believe is acceptable for scheduling measurement tasks. It is unlikely that user-generated activity will create such a large burst of measurements, so we do not expect these delays to affect user-perceived responsiveness.

We similarly measure the scheduling delay introduced by *Mobilyzer*, *i.e.,* the delays in starting the measurement and sending the result (schedule overhead part I and II in Fig. 4.4). The scheduling delay distribution for a single client is shown in Fig. 4.6. It is slightly higher

Task	Power consumption under WiFi (mAh)	Power consumption under LTE (mAh)
DNS	0.03 (0.01)	0.09 (0.01)
HTTP	0.05 (0.01)	0.12 (0.01)
UDP (Down)	0.06 (0.02)	0.17 (0.02)
UDP (Up)	0.08 (0.03)	0.17 (0.04)
PING	0.21 (0.01)	0.52 (0.02)
TCP (Down)	0.11 (0.28)	2.88 (0.24)
TCP (Up)	1.33 (0.04)	2.36 (0.20)
Traceroute	0.75 (0.25)	2.34 (0.02)
Sum(unbatched)	3.51 (0.38)	8.66 (0.32)
Sum(batched)	3.75 (0.61)	5.06 (0.25)

Table 4.4: Power consumption of *Mobilyzer*: each cell lists the average, then the standard deviation in parentheses.

than the IPC overhead, but still on the order of tens of milliseconds. Again, we believe this is acceptable for scheduling and reporting results from measurement tasks.

4.4.3.2 Power Usage

We now estimate the power consumed by *Mobilyzer*. In general, it is difficult to distinguish power consumption of our library from the application that runs it. To address this, we measure the power consumption of an app using the *Mobilyzer* service as it runs in the background executing server scheduled measurement tasks and compare it to power consumed by the same app before integrating *Mobilyzer*, with the overall functionality kept the same.

We measure the power consumption of measurement tasks using a Samsung Galaxy Note 3 as the test device, and a Monsoon power monitor [26] to measure the device power usage.We run each task 5 times with a 15-second delay between tasks to determine the power drain in isolation (unbatched). All experiments are repeated 3 times to identify possible outliers, and we record the average power consumption. Additionally, we run a batched task consisting of all 8 tasks 5 times, and measure the power consumption during the entire measurement period to better reflect that most tasks in *Mobilyzer* run in batches. For comparison, we also sum the average power consumption for all 8 tasks, denoted as

Sum(unbatched) based on individual experiments. Table 4.4 shows the results for WiFi and LTE.

Of the unbatched experiments, we can see that the TCP throughput test has the highest power consumption. This is consistent with prior results showing that power drain is roughly linear with the throughput and duration that the network interface is in a high power state [100]. The power drain from remaining tasks is proportional to the measurement duration because their throughput is low. Note that traceroute has higher power consumption under LTE than under WiFi, since it sends a large number of ICMP packets with an interval of roughly 0.5s. While on LTE, the device will stay in the high power state between these packets.

For batched tasks, we found that on WiFi the power consumption for batched and unbatched tasks are similar. However, on LTE the total power consumption of the batched measurements is much smaller than that of the sum of the individual measurements, by 41.5%. This is because of the tail energy effect on LTE, where the device remains in a high power radio state for several seconds after a network transmission. In each individual task, there is a contribution from the tail time of an average of 48.2% to the total energy consumed [100]. When the tasks are batched, there is only one tail time, demonstrating the effect of batching in saving energy in cellular networks.

These microbenchmarks identify the energy for individual measurements, but not the relative cost compared to other services running on devices. We evaluate this using the built-in Android battery consumption interface on a Samsung Galaxy S4 device actively running a *Mobilyzer*-enabled app with a 250 MB monthly measurement quota. We find that the power consumed by *Mobilyzer* in this scenario is nearly identical to the Android OS itself, with each comprising about 5% of the total energy drain. We believe this power consumption to be reasonably low for most users, and emphasize that *Mobilyzer* allows users to specify limits on the maximum energy our measurements consume.

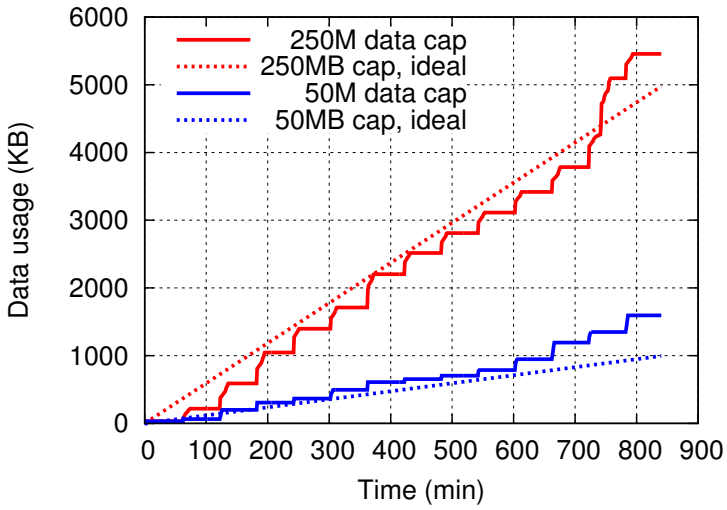


Figure 4.7: Data usage under different data caps.

4.4.3.3 Data Usage

We now evaluate adaptive measurement scheduling in response to per-device caps of *Mobilyzer* data consumption. Our system consumes data quota from (i) fetching measurement requests from the global manager, (ii) uploading measurement results, and (iii) running the measurement tasks. The last category constitutes the vast majority of data consumption, so we set the frequency of periodic measurements according to each device's data cap. We now demonstrate how well this works in practice.

For this experiment, we monitor the data consumption on a HTC One device by reading the proc files under `/proc/uid_stat/<app uid>` once per minute to monitor the app-specific data usage for *Mobilyzer*. Figure 4.7 shows *Mobilyzer*'s data usage on a cellular network during 14 hours for a 250 MB and 50 MB data cap, along with the corresponding ideal data consumption. The server adjusts the measurement frequency for tasks based on estimates of the data consumed by each task and the device's data cap. Data is consumed at a slightly higher rate than expected, as the server estimates are not precise. For more precise data consumption control, a client-side data monitor measures all data consumed by *Mobilyzer* and stops running server-scheduled tasks as soon as the limit is reached.

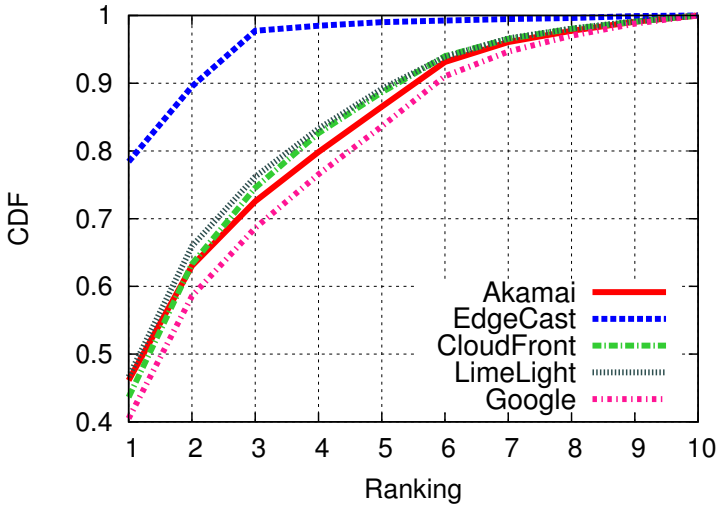


Figure 4.8: CDF of CDN-selected server rank (in terms of latency) compared to ten other CDN servers. WiFi and cellular redirections have similar distributions, so we show only the aggregated result here.

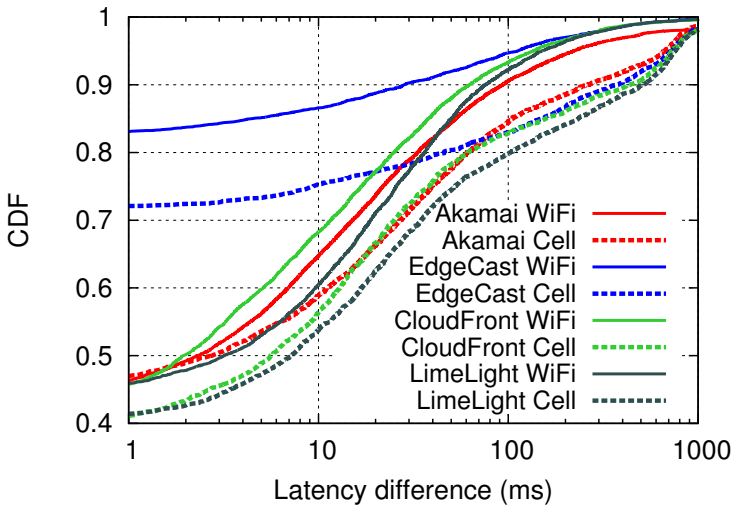


Figure 4.9: CDF of ping latency difference between the CDN-selected server and the CDN IP with lowest latency.

4.4.4 Server Scheduling

The *Mobilyzer* global scheduler enables dynamic, interactive measurement experiments where tasks assigned to devices vary in response to results reported from prior measurements. We present two applications of this feature: measuring the *CDN redirection effectiveness* of major CDNs, and *network diagnosis with adaptive scheduling*, which dynamically schedules diagnosis measurements in response to observed performance issues.

4.4.4.1 CDN Redirection Effectiveness

In previous work [169], Su et al. used PlanetLab-based experiments to show that CDNs typically send clients to replica servers on low latency paths, instead of optimizing for factors such as load balancing. This requires measuring paths not only to replica servers returned by a DNS redirection, but also testing the latency to other replica servers that could have been selected. Using a dynamic measurement experiment, we repeat this study for the mobile environment to test the extent to which this holds true.

Specifically, we select DNS names served by five large CDNs (Akamai, LimeLight, Google, Amazon CloudFront, and EdgeCast) and measure the latency to the servers that mobile clients are directed toward via DNS lookups. In addition, we measure the latency to servers (5 servers with lowest latency from the recent device measurements and 5 randomly selected servers in distinct /24 prefixes) that *other* devices are directed toward.

For each round of measurement, we find the delay difference between the lowest-latency server and the one returned by the DNS lookup to determine the quality of the mapping. In addition, we sort the latencies to determine the *rank* of the server. For example, if the server returned by the CDN is the second-fastest of ten, its rank is 2.

We summarize our rank results in Figure 4.8 for 476 devices which used either WiFi (860K measurements) or cellular (393K measurements). Each data point represents the ranking result for one round of measurement.

The figures show that CDNs do not pick the best server half of the time, and as we show

below, a this leads to a significant performance penalty for many of our measurements. We calculate the latency difference between the lowest-latency CDN replica and the CDN-selected replica for the same devices in each round of ping measurements. In more than 40% of cases, the latency to the CDN-selected server is optimal. Figure 4.9 plots the latency difference for cases where the CDN-selected replica is not optimal. We find that for cellular users, 20% of the cases lead to a latency difference of higher than 100ms, which is particularly bad for small downloads common in Web pages. In the worst 10% of cases, there is a noticeable difference between WiFi and cellular; for some CDNs (*e.g.,* Akamai and EdgeCast), the latency difference for WiFi is 50 ms, compared to 500 ms for cellular. While we do not identify the reason for each of these cases, previous work identifies a variety of cases caused by path inflation [188]. Importantly, in contrast to previous work in 2011 showing little opportunity for CDN optimization in mobile networks [181], our study indicates that mobile carriers have more egress points today and CDNs have multiple options for serving clients, but they do not always make optimal decisions for mapping clients to CDN replicas, which can have a significant impact on performance.

4.4.4.2 Network Diagnosis with Adaptive Scheduling

Dynamic scheduling not only enables new measurements in the mobile environment, it also provides an opportunity to improve measurement efficiency. For example, consider a measurement experiment that measures performance periodically and issues diagnosis measurements in response to anomalous conditions, *e.g.,* to isolate whether there is a problem with a specific server, endpoint or network. Without a priori knowledge about when and where network problems will manifest, the only way to ensure diagnosis information is always available is to issue diagnosis measurements even when there is no problem detected. We now use two examples to show how *Mobilyzer* allows us to issue diagnosis measurements on demand in response to observed problems. For both cases, the global manager scheduled a diagnosis traceroute task only after observing an increase in ping

round trip time.

Data Roaming. When roaming, a subscriber's IP traffic may egress into the public Internet directly from the roaming carrier, or it can be forwarded to the home carrier [44], to facilitate data-usage accounting and provide access to services located only in the home carrier. Our diagnosis measurements identified the cost of the latter approach. When a Verizon (US) user roamed on five Europe carriers, the that latency to google.com (server in US) increased from 198±46ms to 613±34ms for the same cellular access technology. Our traceroutes reveal that the first hop of the path belongs to Verizon network for all the measurements from five roaming carriers and the first hop round trip time increased from 159±47ms (non-roaming) to 469±26ms (roaming), indicating that *traffic from Europe was tunneled to the US and back for this user, incurring hundreds of additional milliseconds of delay*.

Path Dynamics. In this example, the path measurement revealed that 100 ms increase in latency for a Vinaphone user was due to a transient path change (perhaps due to a failure and/or BGP update): latency and hop counts increased by a factor of 3 and 1.5, respectively.

4.5 New Measurements Enabled

Mobilyzer enables us to crowdsource and measure[2] the performance of two popular Internet services on mobile devices: Web page load time (PLT) and Video QoE. We use the PLT measurement to show that mobile CPUs can be a significant bottleneck in browser performance, and simple optimizations can improve it. With our video QoE testing, we compare the performance of different bit-rate adaptation schemes in the mobile environment.

[2]This study is IRB-approved and participating users are consented before participating via an in-app dialog. Users may report data via a Google account, which allows them to view and delete their data at any time, or they may report data anonymously. We strip all user information and IDs from the data we gather; further, we coarsen the location granularity to one square kilometer.

4.5.1 Page Load Time Measurement

Recent studies show that a significant fraction of mobile Internet traffic uses HTTP [158], and many of those flows come from Web browsing. The wait time between requesting a Web page and the page being rendered by the browser has a significant impact on users' quality of experience (QoE). This is often measured using the PLT metric, defined to be the delay to fetch all the resources for a page. In addition to measuring PLT for mobile devices, our work is the first to characterize and compare it with page interactive time (PIT), which is the delay to first render the page by the browser so that a user can interact with it.

PLT and PIT are important performance metrics, useful for predicting user experiences. Large PLT and PIT values can lead to users abandoning the pages. As a result, there are several efforts to improve PLT measurements. Browsers collect and report fine-grained page performance and usage info using the Telemetry API [6, 43], without identifying the reasons for the measured performance. Systems such as WebPageTest [48] and others [61] provide similar information from a small set of dedicated hosts in multiple locations. The Wprof [177] approach helps explain the resource dependencies that affect page load times, but it requires a custom browser and has been used primarily in a lab environment.

Importantly, there is a poor understanding of PLTs in the *mobile environment*. The key challenge is that it is difficult to instrument existing mobile browsers (*e.g.,* because they do not support extensions). Huang et al. [97] studied and measured mobile Web browsing performance using controlled experiments and inferred PLT from packet traces. However, we still have a limited understanding of the key factors that affect PLT behavior in the mobile environment.

In this section, we use *Mobilyzer* to conduct the first crowdsourced measurements of mobile Web page performance. We use a novel methodology that combines empirical data from resource download timings with Wprof dependency graphs generated outside the mobile environment. This allows us not only to measure summary statistics like PLT

and PIT, but also understand bottlenecks such as network and computation.

4.5.1.1 Methodology

We use a PLT measurement task to investigate page load times on mobile devices. Previous methods to measure PLT either instrument the browser to report resource load timings [177], or rely on packet traces to infer the page load time [97, 148]. Some also used the change in layout to infer PLT [68]. Using crowdsourcing, we cannot modify a browser or infer from network traces. Instead, we use the Navigation Timing (NT) API as described below, which is arguably one of the most accurate ways of measuring PLT, as uses system clock time and measured by the browser itself.

Metrics. Our implementation loads a URL in a `WebView`, a browser commonly embedded in mobile apps. We measure the time for a `WebView` to load a particular page using the NT API [27], a W3C standard implemented in all browsers. This allows us to measure timings for DNS lookup, TCP handshake, transfer of the main HTML file, parsing and constructing the Document Object Model (DOM), render time, and the completion of the page load. After loading a URL using `WebView`, we use JavaScript to read these timings and report them in the measurement result.

A key limitation of the NT API is that it reports timings only for the main HTML page, not any other referenced resources. To get resource timings, we intercept resource requests from the `WebView` using an Android API that notifies our PLT task when each resource is requested, allowing us to capture all of a page's resources and their timings.

When parsing completes and the DOM is constructed, the browser can start rendering and painting the initial view of the page. At this time, called *above-the-fold render time*, page becomes interactive. In this chapter, we refer to this as the *page interactive time* (PIT). Developers argue that PIT is a better metric to quantify the user perceived performance of the page than the total PLT, as it can reflect how quickly the user can begin interacting with the page [89].

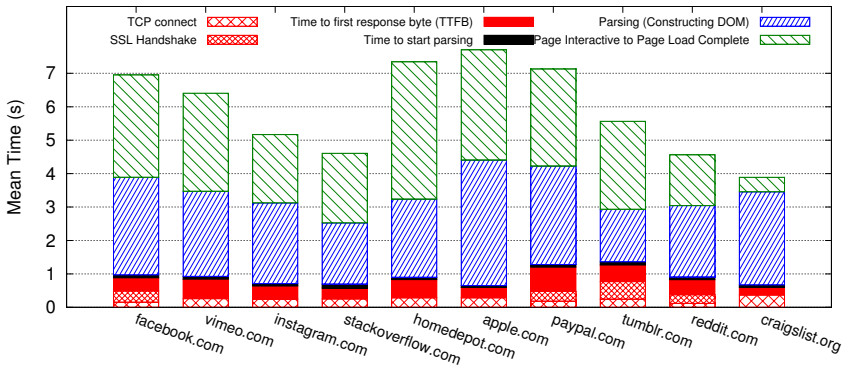


Figure 4.10: Navigation timing data from crowdsourced PLT measurements, comprising 80 users during six weeks (2000 measurements per page on average).

Pages measured. To study the PLT of webpages on mobile devices, we selected ten popular webpages, one from each of the following categories in the top 100 Alexa pages: social networking, image and video hosting services, forums, wikis, shopping, and banking Web sites. For sites with a landing page that only shows a login screen or redirects the client to an Android app, we selected a custom profile page for testing to obtain a more meaningful result.

4.5.1.2 Summary results

We collected measurements from 80 mobile devices worldwide, providing resource timings for 2,000 measurements per page on average. We present summary results in Fig. 4.10, which separates page load time into TCP handshake, SSL handshake, time to first byte, parsing, PIT, and total PLT.

A key result is that total PLTs range from 4 (craiglist) to 7 (apple.com) seconds, which is substantially larger than reported in desktop environments (*e.g.,* most PLTs were less than 2 seconds in a previous study [177]). As expected, the pages with higher total PLTs tend to be more resource-rich (images, JavaScript). Fig. 4.10 shows that the initial network costs (TCP and SSL handshake, time to first byte) are a small fraction of total page load time. That said, the SSL handshake can add up to hundreds of milliseconds of delay in

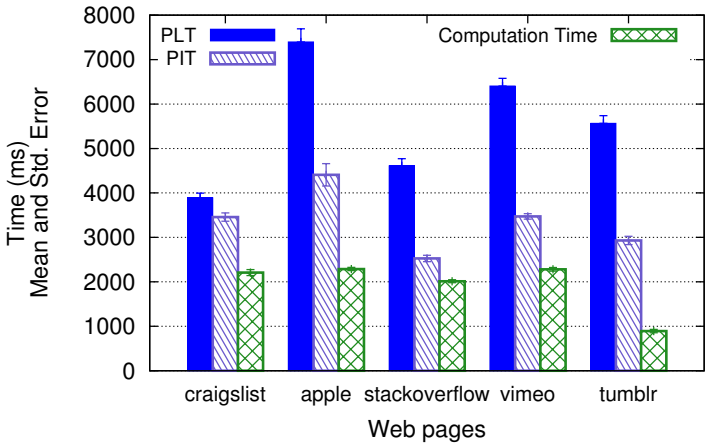


Figure 4.11: Browser computation time versus PLT and PIT. The fraction of load time due to computation is variable, but often a significant portion.

the PLT [75]. Importantly, the network delays for loading pages are nearly identical for all pages and quite small, suggesting that *improving network latencies alone is unlikely to significantly reduce PLTs in the mobile environment*.

The figure also shows that most of the page load time comes from rendering and resource loading. It is important to note the large gap between PIT (when the DOM construction is complete) and total PLT. This typically occurs due to network delays for downloading resources and processing delays for parsing them. In the next section, we disentangle these two factors to identify the bottlenecks for page rendering on mobile devices.

Based on the gap between PIT and PLT, the proportion of page interactive time to total page load time ranges from 45% for homedepot to 90% for craigslist, and the mean ratio of PIT to PLT is 0.62 (median 0.63) for the top 200 Alexa pages. This is in contrast to findings in previous work [148], which reported that for 60% of the pages, page render time is equal to page load time. Our results indicate that *it is important to consider both PIT and PLT because of their high variability across sites*.

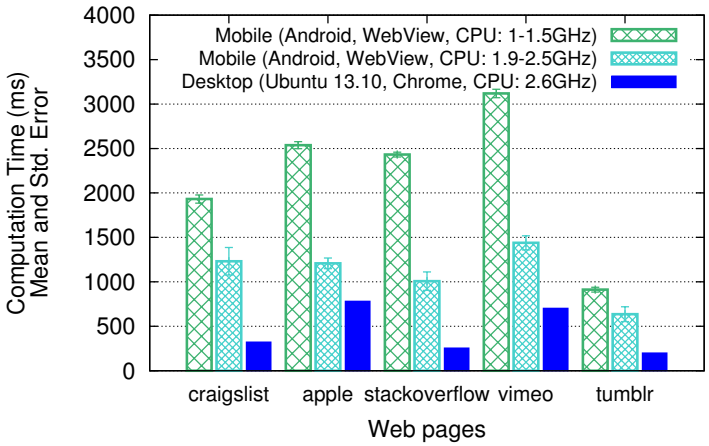


Figure 4.12: Computation time of different mobile processors (30 distinct device models) compared with desktop.

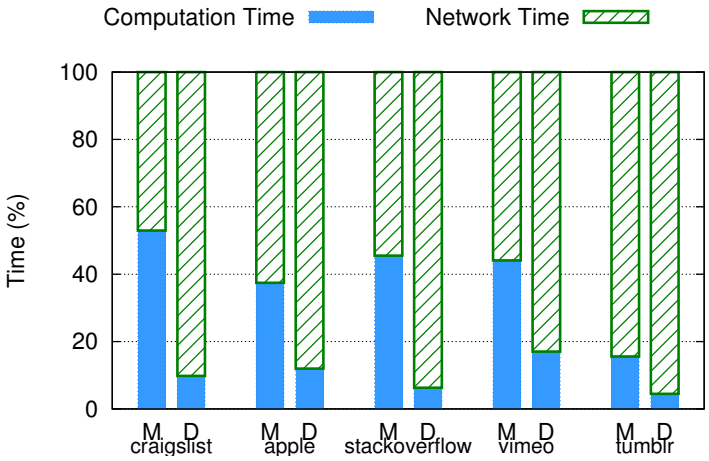


Figure 4.13: Computation time in mobile devices can account for more than half of the PLT. (Desktop was connected to a 3Mbps broadband internet service.)

4.5.1.3 Is the bottleneck computation or network?

The previous analysis describes that most of the user delays for loading Web pages result from parsing and resource loading, without further separating the time spent downloading objects from that processing them. We now disentangle these factors and demonstrate that processing delays are by far the largest bottleneck.

Inferring processing time. This requires understanding when a browser spends time downloading objects versus processing them — the NT API reports only the start the the end of the resource download, without detailing the processing time. The Wprof project

provides this missing information, but requires running a custom browser that we could not distribute to our users. We propose a new methodology consisting of running Wprof analysis on a Web page loaded in our lab environment, followed by using the resulting dependency graph to infer processing times based on empirical resource timing gathered using *Mobilyzer*.

We argue that the dependency graph generated by Wprof for Chrome is likely identical to the one for the Android WebView, because of these reasons: (1) most of the dependencies in the pages are flow dependencies, which are imposed by all browsers [177], and (2) both Android WebView and Chrome use WebKit as their rendering engine. To help validate this assumption, we compared the download order of resources in Chrome (dependency graph) and Android WebView (resource timing), and found them to be consistent.

Results. To determine the amount of computation time, we exclude the network transfer time; *i.e.,* we emulate the case where all resources are cached. Figure 4.13 presents the average and standard deviation for PLT, PIT and computation time. The results show that for most of the pages in our study, computation is responsible for about half of the PLT and an even larger fraction of PIT. Note that for pages such as Tumblr with fewer scripts or more static content (*e.g.,* images), the network is responsible for approximately 80% of total PLT. To compare, the Wprof study reported the median portion of computation time for the top 200 Alexa pages in the desktop environment to be 35%, higher than the computation time of our five pages (Fig. 4.13). This difference results from using the mobile version of pages in our experiments, which are simpler than the full version used in previous work.

We further used our crowdsourced measurements to understand the impact of mobile-device CPU speed on loading Web pages. We grouped the devices in our dataset into two clock frequency ranges of their processor: 1-1.5 GHz (17 distinct models) and 1.9-2.5 GHz (13 distinct models). We also loaded the pages on a desktop computer with a 2.6GHz CPU to compare the computation time in a device with a powerful processor. Figure 4.12 demonstrates that PLT in mobile devices can significantly benefit from faster

CPUs. Specifically, the computation time of smartphones with 1.9-2.5GHz processor is around half of the smartphones with 1-1.5GHz processors on average. *Thus, Web site load times can significantly benefit from faster processors*.

Importantly, desktop load times are significantly faster than mobile devices. A potential pitfall of using desktops to simulate the mobile environment is clear: while Wprof finds that increasing the CPU frequency from 2GHz to 3GHz will decrease the computation time by only 5%, we find in the mobile environment that the benefit of using faster processors is significantly higher. Figure 4.13 indicates that, compared to a desktop computer with a 3 Mbps downlink (*i.e.,* similar to 3G's 2-3 Mbps), the CPU resources in smartphones is a bottleneck for Web page rendering, as it accounts for half of the total PLT.

4.5.1.4 Critical Rendering Path and Render Time

To shorten the PIT, developers attempt to optimize the critical rendering path (CRP) [14] — the sequence of steps the browser uses to construct the DOM and render the initial view of a Web page. During parsing, some network and computation processes can occur in parallel, but certain blocking resources prevent parallelism and are the main factors for the CRP time. Importantly, optimizing resources not on the CRP (*non-blocking* resources) cannot shorten the render time. We now use our Wprof-generated dependency graphs and empirical page load times to diagnose cases where the CRP is unduly long.

We compare the total parsing time and total computation time in Figures 4.10 and 4.11, and find that for some of the pages, total computation time differs from the total parsing time, implying that the parser is blocked by the network. For example, for craigslist and tumblr, the blocking time due to network in the CRP comprises 27% and 24% of total parsing time, respectively. By investigating the dependency graphs for tumblr and craigslist, we find that large blocking JavaScript files (150-260KB) are the culprit, since these resources may not have been downloaded yet when the parser needs to process them. In some cases, these problems can easily be remediated: for craiglist the 255 KB JavaScript file would

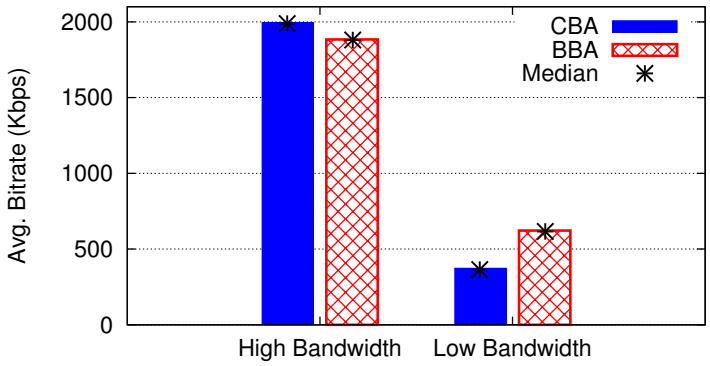


Figure 4.14: Comparing BBA with CBA bitrates and throughput for users with throughputs that are higher and lower than the maximum bitrate.

require only 77 KB if the server supported gzip compression.

4.5.2 Video QoE Measurement

Video streaming over mobile devices is gaining popularity: YouTube [16] reorted recently that about 50% of views come from a mobile phone or tablet. Further, the average bandwidth achievable in cellular networks in 2014 is 11 Mbps [34], sufficient for high resolution content (1440p).

Despite the higher capacities in today's mobile networks, a key challenge for streaming video providers is how to ensure reliable and high quality video delivery in the face of variable network conditions common in the mobile environment. This requires avoiding stalls and rebuffering events, and adapting the video bitrate to available network bandwidth. This is commonly done via Dynamic Adaptive Streaming over HTTP (DASH), which tries to seamlessly adapt the requested video quality based on available bandwidth, device capability and other factors. In this section, we evaluate the relative performance of two alternative schemes for DASH, using crowdsourced measurements over mobile networks experiencing a diversity of network conditions.

Methodology. We use the ExoPlayer library [12], which provides a pre-built customizable video player for Android with DASH. ExoPlayer is currently used by YouTube and Google

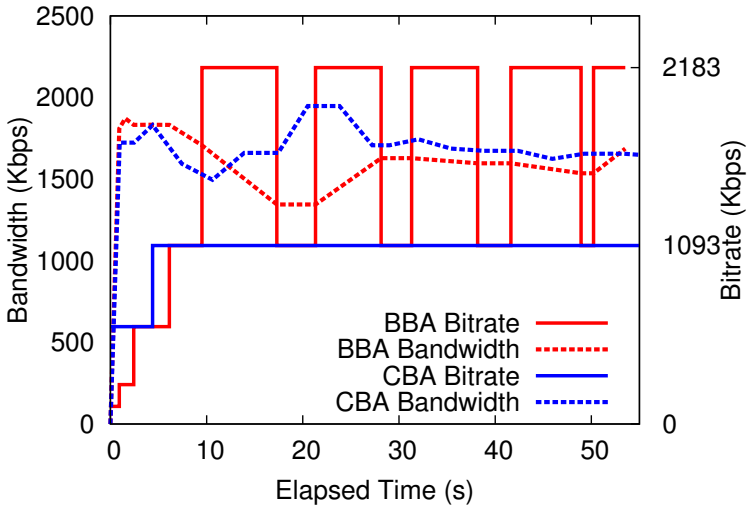


Figure 4.15: Timeline plots of typical throughputs and bitrate adaptations for BBA and CBA. BBA tends to achieve higher average bitrates than CBA with minimal impact on rebuffering.

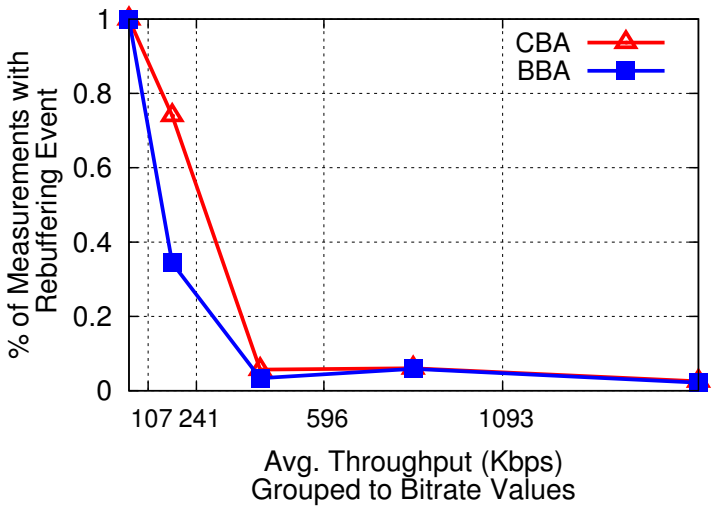


Figure 4.16: BBA leads to lower rebuffering rates compared with CBA, particularly for users with lower throughput.

Play Movies [13], thus allowing us to directly apply our findings to these popular video services. This library allows us to record throughput, bitrates and rebuffering events over time during video playback.

Using ExoPlayer, we implement a recently proposed buffer-based adaptive (BBA) streaming algorithm [98], and compare it with capacity-based adaptive (CBA) streaming, implemented by YouTube. Both CBA and BBA attempt to minimize rebuffering during playback. BBA [98] dynamically adapts the video bitrate by monitoring the buffer occupancy. To avoid rebuffering, it downloads chunks with a lower bitrate when the buffer starts draining. During the startup phase, it uses the estimated bandwidth to select the appropriate bitrate. On the other hand, CBA only considers estimated bandwidth when selecting the proper bitrate. It always chooses a bitrate that is less than or equal to the estimated bandwidth.

We use a two-minute YouTube video[3] with 5 bitrates (144p to 720p) for our measurements, and streamed it using BBA and CBA on *Mobilyzer* clients. We collected 10K video measurements from 40 users during two weeks.

Results. We considered two QoE metrics to evaluate the performance of these algorithms: (1) Average bitrate of the video that is shown to user, which shows the average quality of video, and (2) rebuffering events, which reports if the video gets interrupted during the playback.

Average Bitrate. To investigate the performance of BBA and CBA under different conditions, we group the measurements based on their average bandwidth[4] into two groups: larger than highest video bitrate ($2193Kbps$) and smaller than the highest bitrate. In the former case, there should be no need for rate adaptation, and in the latter, the streaming algorithm must pick a bitrate lower than the highest quality.

Figure 4.14 shows the average throughput and bitrates achieved by the two algorithms

[3]We pick this value to limit data consumption; further, a recent study indicates that most video views are less than 3 minutes [35].

[4]Bandwidth is estimated from the chunk download times.

in crowdsourced measurements. In the low bandwidth scenario, BBA displays videos with a higher average bitrate. To understand why, we focus on two BBA and CBA experiments with the same throughput during the measurement (Fig. 4.15). The figure shows that BBA switches to a higher bitrate when the buffer occupancy is high, while CBA does not adapt its rate because it considers only the instantaneous estimated throughput. Therefore, when available bandwidth falls between available bitrates (or switches between them), BBA will provide a higher bitrate than CBA on average. On the other hand, we observe that CBA adapts more quickly to sudden changes in throughput by switching to higher or lower bitrates, while BBA maintains its current bitrate based on its current buffer occupancy.

For devices with larger bandwidth, CBA provides a slightly higher average bitrate than BBA. This occurs because BBA is more conservative and increases its bitrate incrementally during the startup phase.

Rebuffering. A significant additional component of video QoE is the rate of rebuffering events — the lower, the better. We investigate the performance of BBA and CBA using this metric in Figure 4.16, which shows the percentage of measurements with rebuffering events. We group our results into five bandwidth buckets based on the video bitrates. The figure shows that when the average bandwidth is lower than the smallest throughput, all the measurements experience at least some rebuffering. However, with higher bandwidths, BBA experiences fewer rebuffering events. This occurs because it avoids rebuffering by switching to the lowest bitrate when the buffer is nearly drained.

Competition under limited bandwidth. To understand how BBA and CBA behave when they compete for a limited bandwidth, we stream videos using a smartphone connected to an AP with constrained bandwidth. We played a BBA and CBA stream at the same time using a *Mobilyzer* parallel task. When using the *load control* ExoPlayer feature, streaming pauses when the playback buffer reaches a threshold; in this case, CBA will pauses for 22.9s on average, while BBA pauses only for 2.9s. This lower pause time occurs because BBA switches to higher bitrates when the buffer is nearly full. Thus, while the throughput

is lower than the highest bitrate, BBA will not pause. In contrast, playback buffer size for CBA will grow continuously, due to choosing a bitrate lower than the estimated throughput.

These behaviors have a significant impact on energy and performance. In terms of energy, CBA improves efficiency by letting the radio go idle during pauses. However, when CBA pauses, BBA consumes the remaining bandwidth and achieves higher average throughput. In our experiments, BBA achieved 63% higher throughput and 57% higher bitrate than CBA, meaning that BBA will provide higher QoE.

In summary, our evaluation shows that the BBA algorithm performs better than CBA in the mobile environment based on common QoE metrics. Further investigation is needed to determine how frequent rate adaptation affects the user experience, and to understand the broader network impact of large numbers of clients using BBA instead of CBA.

4.6 Conclusion

This project makes the case for network measurement as a service in the mobile environment, and demonstrates how we addressed several challenges toward making this service practical and efficient. *Mobilyzer* provides (i) network isolation to ensure valid measurement results, (ii) contextual information to ensure proper scheduling of measurements and interpretation of results, (iii) a global view of available resources to facilitate dynamic, distributed experiments and (iv) a deployment model with incentives for experimenters and app developers. We showed that our system is efficient, easy to use and supports new measurement experiments in the mobile environment. As part of our future work, we are designing an interface to facilitate new measurement experiments and testing new ideas for predicting resource availability and optimizing measurement deployment.

CHAPTER V

An In-depth Understanding of Mobile Multipath: Measurement and System Design

5.1 Introduction

The support for multiple network interfaces is a norm on today's smart devices: smartphones and tablets often have both WiFi and cellular connectivity; wearable devices are capable of pairing with their phones using either Bluetooth or Direct WiFi; even Internet-of-Things (IoT) devices such as home alarms and smoke detectors can potentially leverage multipath for traffic offloading [176].

Multipath provides new opportunities for improving mobile application performance. The most widely used multipath solution is Multipath TCP (MPTCP) [85], which allows unmodified applications to transfer data over multipath. In the research community, numerous studies have been conducted on multipath [70, 77, 130, 129, 71, 140, 92, 172, 93]. Industry has also been enthusiastically adopting multipath. Some built-in apps in iOS, *e.g.,* Siri, support multipath [21]. A Korean R&D Center plans to commercialize "GiGA LTE", which achieves Gbps throughput on commodity smartphones over multipath [22].

Despite these efforts, there still remain numerous challenges for effectively and efficiently using mobile multipath. To name a few, first, originally designed for data center networking [151], MPTCP may incur unexpected cross-layer interactions when used on

mobile devices [92, 77]. Second, as shown later, MPTCP often incurs additional energy overhead, but may not always boost (sometimes even worsen) application performance. Therefore, applications should use multipath only when its benefit outweighs its incurred overhead. Such decision logic is largely missing or done naïvely in practice. Third, protocols such as MPTCP are complex with numerous configurations, e.g., scheduling, and it is unclear how to tune them in an optimal way. Fourth, from a system architectural perspective, MPTCP's "everything in kernel" scheme may not be suitable for mobile multipath to support a rich set of application-specific policies.

In this chapter, we conduct an in-depth measurement-driven study of multipath over mobile devices, with the goal of providing key knowledge and vital clues for evolving the mobile multipath design. More specifically, we explore the following unanswered questions.

• *How much performance benefits can MPTCP offer in real-world settings?* Differing from prior studies [70, 77, 130, 92] performing in-lab controlled experiments, we launch an IRB-approved user trial by collecting network traces from real users' smartphones with MPTCP enabled. Meanwhile, we complement the passive measurements with crowdsourced active probing on participants' phones, to capture application quality of experience (QoE) over multipath under diverse network conditions. To our knowledge, this is the most in-depth user study of mobile multipath that focuses not only on multipath itself, but also on cross-layer interactions. Leveraging the unique data we collected, we analyzed MPTCP behaviors "in the wild", including multipath availability, path utilization, handshake latency, TCP throughput, web page load time, video streaming bitrate. We also examined multipath for voice-over-IP (VoIP) and instance messenger (IM) whose multipath-friendliness have not been explored before. From the user study, we found that multipath is widely available and it incurs rather complex interactions with applications due to their diverse traffic patterns and QoE metrics. MPTCP's suboptimal performance often stems from three factors: short flow duration, excessive delay of subflows' handshakes, and the scheduling

algorithms.

• *What is the energy footprint of MPTCP on real users?* MPTCP incurs additional energy overhead [172, 130]. We analyzed this overhead by applying widely previously validated single-path and multipath radio energy models on real users' network traffic. We found that properly using multipath incurs reasonable and manageable overhead. However, blindly applying MPTCP to all traffic, as is usually done today, increases users' average radio energy by 1.08X. Our study shows that many optimizations can be performed to reduce multipath's energy footprint, such as improving the scheduling algorithm for small transfers and disabling multipath for applications such as VoIP and IM.

• *How to improve the interplay between multipath and server selection?* We consider a classic problem of CDN server selection. We found that under multipath, the state-of-the-art DNS-based server selection often leads to suboptimal server selection. This is attributed to the fact that CDNs' DNS infrastructure is unaware of other subflows available to the client. Compared to single-path, under multipath, choosing different CDN servers incurs more complex tradeoffs, and can lead to different network performance depending on MPTCP's scheduling algorithm, traffic patterns, and path characteristics. We are the first to identify the problem of CDN server selection under multipath. We demonstrate its real-world importance by probing Alexa top-500 websites and provide recommendations for multipath-aware server selection.

• *How to improve the system architecture for mobile multipath?* The measurement results indicate that MPTCP suffers from a few limitations: poor interaction with short/small flows, a lack of infrastructural support for multipath policy, and MPTCP extension often blocked by middleboxes. We propose a flexible software architecture of mobile multipath called MPFlex that overcomes all the above limitations. MPFlex has several prominent features. First, it performs transparent *multiplexing* for application traffic over multipath. Our multiplexing scheme reduces the number of handshakes from many (one per path) to zero, leading to significant improvement of bandwidth utilization for small flows. Second,

Table 5.1: Summary of the findings and proposed improvement

Measurement Finding	**Proposed Improvement**
Inefficiencies for small flows (§5.3.1): (1) low path utilization for the secondary path, (2) low MPTCP throughput, (3) excessive energy consumption.	Multiplexing and zero-RTT multipath connection establishment (§5.4.2).
Performance and resource impact of MPTCP on different applications differs. (§5.3.1, §5.3.2, and §5.3.3)	A framework supporting multipath policy (§5.4.2).
Sub-optimal CDN server selection for MPTCP connections (§5.3.4).	Multipath-aware CDN selection (§5.3.4).

MPFlex *decouples* the high-level scheduling algorithm and the low-level OS protocol implementation. This is realized by maintaining most of MPFlex's logic in the user space, which obtains lower-layer information (*e.g.,* latency and congestion window) from kernel through a unified API. Such a framework dramatically simplifies the development, deployment, and maintenance of multipath features. Third, MPFlex has visibility of all traffic on an end host, and thus provides an ideal vantage point for applying user-specified multipath policies. Fourth, MPFlex is middlebox-friendly as it does not use any Layer 3 or 4 protocol extensions which may be blocked by ISPs. Compared to MPTCP, MPFlex reduces single file transfer time by up to 49%, improves bundled short flows' transfer time by up to 63%, and boosts real web page load speed by up to 20%, while incurring negligible overhead. We also demonstrate MPFlex's capability of flexibly plugging-in new features such as buffer-aware scheduling, smart packet reinjection, and per-application policies, which can be implemented in less than 70 lines of user-level code.

Overall, we make key contributions to mobile multipath research in two aspects: crowdsourced measurement (§5.2 and §5.3) and improved software architecture (§5.4 and §5.5).

5.2 Measurement Methodology

We describe our IRB-approved user trial to reveal characteristics of mobile multipath traffic "in the wild".

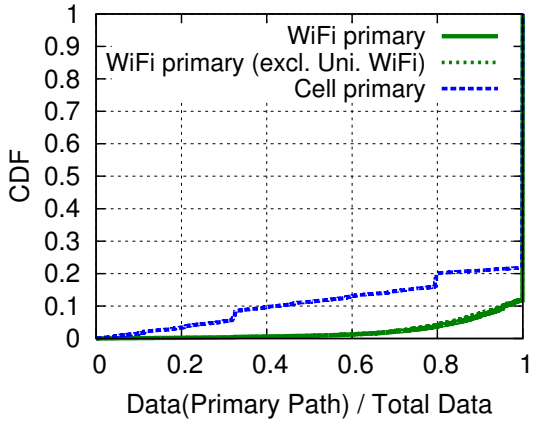


Figure 5.1: Distributions of the fractions of payload transmitted over the primary subflow, across all MPTCP flows.

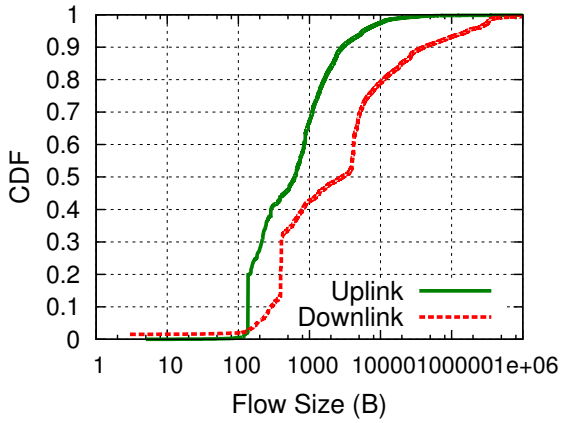


Figure 5.2: Distributions of DL/UL bytes in a SPTCP/MPTCP flow.

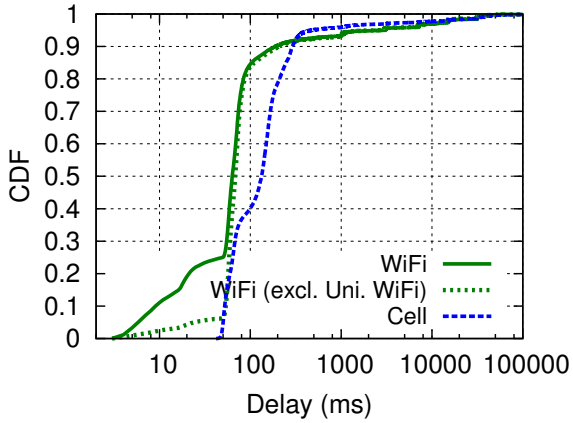


Figure 5.3: Distributions of the handshake delays of MPTCP secondary subflow.

5.2.1 Multipath Configuration For User Trial

We consider a common usage scenario where WiFi and cellular are used at the same time on a smartphone. To realize this, we used MPTCP, the most popular multipath solution with off-the-shelf Linux kernel implementation [138]. We port MPTCP v0.86 to Android 4.4.4 with CyanogenMod 11 ROM on Samsung Galaxy S3 (SGS3) smartphones. MPTCP enables all applications to transparently utilize multipath. However, one challenge is most today's servers do not yet support MPTCP. We thus set up a multipath proxy running MPTCP v0.90. To redirect traffic to the MPTCP proxy, we transparently tunnel all user traffic using Socks5 proxying (using **`shadowsocks`** [40]). The Socks5 protocol [117] only adds a very small header to each packet so its impact on the traffic pattern is negligible. We also verified Socks5 incurs very small runtime overhead.

We recruited 15 students studying at our University and gave each a SGS3 smartphone with unlimited cellular data plan of a commercial U.S. cellular carrier. The participants were asked to use the phone normally with WiFi and cellular enabled. We expect multipath is available at least on campus, at participants' home, and in places with free public WiFi.

The proxy is connected to our campus network. This makes the latency of the WiFi path small when participants use the phones on campus. In our data, only 14% of the traffic belong to this case. We separate out such traffic in some of our analysis. We also systematically study the impact of the proxy location in §5.5.4.

5.2.2 User Trial Data Collection

We built a custom data collector running on users' devices. It transparently performs three tasks: passive measurements, active measurements, and data upload. The collector incurs small overhead without noticeable degradation of users' experience, based on our lab testing.

Table 5.2: Statistics of the user study dataset.

Passive Measurements			Active Measurements		
Apps	All TCP Connections*	MPTCP Conns.	Page loads	Video plays	File DLs
122	1516794	441422	24668	3071	4371

* We use "connection" or "flow" interchangeably when referring to a SPTCP/MPTCP connection, and use "subflow" for MPTCP subflow.

Passive Measurements. The data collector passively collects network packet traces (only TCP/IP headers) via a modified `tcpdump` on users' phones (we did not capture traces at the proxy because for some analysis such as energy, we must use client-side traces). Also, the collector re-configures MPTCP settings of its device every 24 hours by randomly selecting one of the three configurations: (1) MPTCP is disabled, (2) MPTCP is enabled with WiFi selected as the primary path, and (3) MPTCP is enabled with cellular as the primary path. Doing so allows us to statistically compare MPTCP and SPTCP (single-path TCP), as well as to study the impact of the primary path.

Active Measurements. To complement passive measurements, the collector periodically (every hour) runs the following active measurements back-to-back in background: (1) stream a 2-min YouTube video; (2) download a file over single- and multipath; (3) load five popular webpages using different configurations of SPTCP and MPTCP. We detail the measurement methodologies in §5.3.2. The collector records key performance metrics such as video QoE (*i.e.,* video quality and rebuffering events), and page load time, which are difficult to obtain from passive measurements. Note the active measurements are only conducted when the screen is off, the battery life permits, and the phone is idle, so it does not impact user experience.

Data Upload. The collected data for both passive and active measurements is uploaded to our measurement server over every day at late night when the phone is charging. We

collected the data for more than 4 months (33 weeks) from October 25, 2015 from the 15 participants, with general statistics shown in Table 5.2.

Controlled In-lab Experiments. Besides the user trial, we also conduct in-lab experiments, which serve two purposes: validating our findings/systems, and covering experiments that are too difficult to conduct in the user trial. Unless otherwise specified, we used a Nexus 5 phone with Android 4.4.4 and MPTCP v0.89.5 for our in-lab experiments conducted in §5.3.3 and §5.3.4.

5.3 Measurement Results

We now present our measurement results summarized in Table 5.1 for both user study and controlled measurement.

5.3.1 Passive Measurements of User Study

Multipath availability. The first question to answer is, how often is multipath available? During the data collection period, for 82% and 40% of the time, users have WiFi and cellular connectivity, respectively. Given cellular (WiFi) path is available, for 73% (42%) of the time, users can leverage both paths. Overall, the results indicate that multipath is common on mobile devices.

Path Utilization. When multipath is available, how effectively are the two paths utilized? Figure 5.1 plots the distributions of the fraction of payload (both uplink and downlink) transmitted over the primary subflow, across all MPTCP connections. When WiFi is used as the primary path, more than 89% of the flows are fully delivered over WiFi. This is attributed to two reasons. First, the vast majority of the flows have small sizes. As shown in Figure 5.2, the 75% percentiles of uplink and downlink bytes within a TCP flow (single- or multipath) are only 1.3KB and 6.7KB, respectively. Second, MPTCP by default performs

subflow handshakes sequentially *i.e.,* handshake of the secondary subflow occurs after the completion of the handshake or the primary subflow. As a result, for short WiFi-primary flows, often the user data can be fully delivered over WiFi before the LTE subflow is established.

Figure 5.3 plots distributions of the handshake delay of the secondary subflows. The median handshake delays of cellular and WiFi secondary subflows are 133ms and 63ms, respectively. As a result, when LTE becomes the primary path, because the WiFi path is established quicker (compared to the LTE path establishment when WiFi is the primary path), there are more opportunities for WiFi to be utilized, as shown in Figure 5.1. Excluding flows whose primary paths connect to the university WiFi (they account for only 11% of the total flows) slightly shifts the low-end of the distribution.

On the other hand, as the flow size increases, the fraction of primary-path bytes decreases. This is because the longer duration for large flows allow both paths to be established and transfer non-trivial amount of data. For all MPTCP flows, 69% of the bytes are transferred over WiFi and 31% are over cellular. A large fraction of these bytes are contributed by a small fraction of large flows due to the heavy-tail distribution of flow sizes [144].

Also note MPTCP can be configured to perform both handshakes simultaneously. This can improve the overall throughput due to shorter handshake time, but it incurs energy overhead, and may bring limited improvement when one path has long latency. We propose a scheme in §5.4 that boosts the multipath performance by reducing the number of handshakes from many to one.

MPTCP Performance. How much performance benefits does MPTCP provide? Here we focus on a simple metric – the transport-layer throughput, and study application QoE later in §5.3.2 and §5.3.3. Figure 5.4 studies five flow size groups within each four schemes are compared: SPTCP over cellular, SPTCP over WiFi, cellular-primary MPTCP, and WiFi-

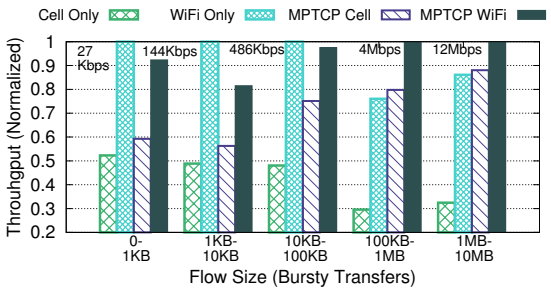


Figure 5.4: Average TCP throughput for different flow size groups.

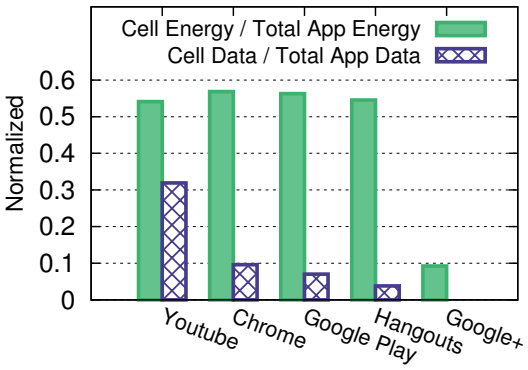


Figure 5.5: Fraction of data delivered by cellular versus the fraction of energy consumed by cellular for popular apps.

primary MPTCP. We normalize the measured throughput at a per-group basis with the maximum throughput in each group (normalized to 1) annotated. Also note for Figure 5.4, we ignore all flows with inter-packet-arrival time greater than 1s to ensure the flow is not likely to be throttled by application.

We make two observations from Figure 5.4. First, for small flows (less than 100KB), SPTCP over WiFi provides good throughput that usually outperforms MPTCP. This is because latency plays a more important role in determining small flows' performance than bandwidth does, and in our dataset WiFi usually has a smaller RTT than cellular so transferring the entire flow over WiFi may provide better performance. A question one may ask is, given MPTCP's scheduler selection the path with the smallest RTT, why does MPTCP not perform at least as well as SPTCP? The reason is, the above path selection only happens when both paths have spaces in congestion window (cwnd). If, for example, WiFi's cwnd

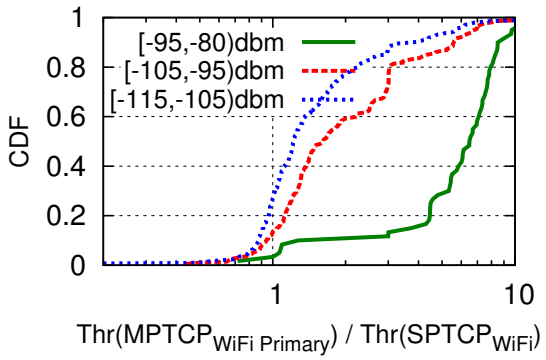


Figure 5.6: Throughput ratio of MPTCP (WiFi primary) to SPTCP over WiFi for 4,371 back-to-back throughput measurements for different value of LTE signal strength.

is full but LTE has empty cwnd space, the data will still be transferred over LTE even if its latency is higher. This explains why MPTCP may incur worse performance than SPTCP does. We will improve this in §5.5.

The second observation from Figure 5.4 relates to large file download where throughput is more important than latency. In this case MPTCP always achieves higher throughput than SPTCP does. Here we see the selection of the primary path still matters even for large files. WiFi-primary MPTCP outperforms cellular-primary because, as mentioned before, the long subflow handshake time of LTE prolongs the overall download time.

Radio Energy is the energy consumed by the radio interface. It accounts for a significant fraction of the overall mobile device energy consumption in particular for cellular (1/3 to 1/2 for 3G [147] and at least 50% for LTE [130]). Multipath impacts the radio energy consumption in two ways. First, it obviously incurs additional energy footprint by using multiple interfaces. Second, if properly leveraged, it may reduce the energy consumption by shortening the file transfer time. In reality, how energy (in)efficient is MPTCP compared to SPTCP? To answer this, we feed the collected trace using the LTE/WiFi radio energy models developed by Nika *et al.* [130] for SGS3, and compute the radio energy consumption. Recall that in the user study, for each user, the data collector re-configures its multipath setting every 24 hours. We found that for MPTCP configurations, their aver-

age radio power is 2.08 times of that for the SPTCP configuration. Based on this, we can roughly estimate the impact of multipath on the overall device battery drain. Assuming for single-path, the radio power (either WiFi or cellular) accounts for 25% of the overall device energy consumption [69], enabling *system-wide* MPTCP shortens the battery life by 21%. We admit though this is a coarse-grained estimation, and it is an upper bound that does not consider the reduced radio energy due to MPTCP's shorter transfer time. We believe this overhead is not unreasonably high, and expect it to be further reduced by using energy-aware policies described later.

Mobile Apps over MPTCP . We pick the five most-heavily used apps by the 15 users over MPTCP to study the resource impact of the cellular subflow. For each app, we plot the fraction of cellular bytes and cellular radio energy. For four out of the five apps, less than 10% of the bytes are delivered over cellular (due to the small flow sizes as explained earlier), while the cellular energy accounts for more than 50% of the total radio energy consumption. For most small flows using WiFi-primary MPTCP, additional energy penalty comes from the fact that even if LTE carries no user payload, the LTE interface still needs to be activated for handshake. This occurs in 91% of the flows with less than 100KB. For these flows, 63% of the radio energy is consumed by cellular that does not deliver any user payload. The energy efficiency increases with larger flows, *e.g.,* for YouTube.

Overall, our findings suggest that although multipath is energy-wise expensive, by strategically leveraging it in an energy-aware manner, the energy overhead can be reasonable and manageable. Some recent work such as eMPTCP [172] has made a first step toward this goal. But eMPTCP suffers from limitations such as reduced throughput (approximately 22% slower on average than MPTCP). We believe that besides improving the scheduler [172], another important premise for energy-aware multipath is to have per-application policy, as different apps incur different cost-benefit tradeoffs when using multipath. We study this in more depth in §5.3.2 and §5.3.3, and propose a system to allow

easy deployment of user policies (§5.4).

5.3.2 Active Measurements of User Study

We now shift our focus to the crowd-sourced active measurement results. Complementing the passive measurement, active measurements provide insights of how multipath impacts the application performance.

File download. We collected 4,371 measurements of downloading a 10MB file using single- and multipath. MPTCP improves the file download performance by 34% in average. Note that the MPTCP throughput is smaller than the sum of both paths' throughput. We found this is because MPTCP's short flow duration makes the congestion window not fully expanded, compared to SPTCP flows.

We next study the MPTCP performance under diverse wireless link qualities. Figure 5.6 plots the ratio between throughput of MPTCP and SPTCP (WiFi-primary), whose measurements were conducted back-to-back, under three ranges of LTE signal strength. Ideally, MPTCP should be at least as good as SPTCP. Surprisingly, when LTE signal strength is poor, for 20% of downloads, SPTCP provides higher throughput than MPTCP. We found this is due to the limited receive window size. Compared to SPTCP, MPTCP requires a larger (connection-level) receive window buffer to absorb out-of-order arrival from multiple paths in particular when the paths have diverse qualities. The default upper limit of the receive window is set too small on Android. We confirmed this through controlled experiments (good WiFi quality, -116dbm LTE signal strength), and observed a strong correlation (Pearson coefficient of 0.79) between MPTCP instantaneous throughput and available receive window space, indicating that decrease in available receiver window causes the sender to slow down the rate on both paths. When we increase its upper limit from the default value (1MB) by 1% (10KB), we observe 12% reduction in download time.

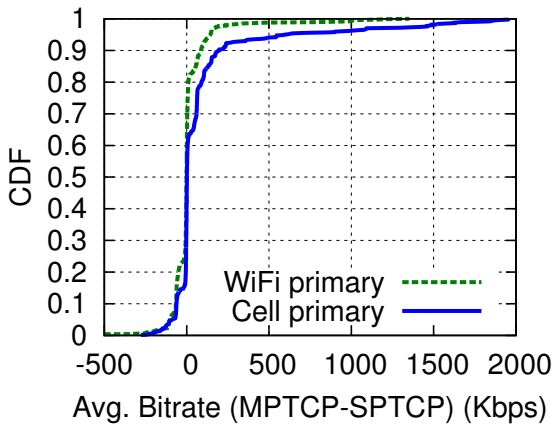


Figure 5.7: Crowd-sourced video streaming measurements (1,500 measurements in total)

Web browsing. We obtained 24K measurements of web page loadings, across five popular sites. We found MPTCP provides small improvement of page load time compared to the best SPTCP result (improvement ranging from 1% to 7%). Inline with prior in-lab measurement on laptops [92], it is attributed to HTTP's traffic pattern of multiple short-lived connections that leave small chances for the secondary subflow to be used. This problem can potentially be mitigated by HTTP/2 that uses multiplexing. But using an HTTP/2 proxy incurs other limitations as to be discussed in §5.4 where we describe our transport-layer multiplexing proposal over multipath.

Video Streaming. We conducted about 3,000 crowd-sourced measurements of capacity-based adaptive (CBA) video streaming, by playing a 2:15 YouTube video with four available bitrates: 240p, 360p, 480p, and 720p. The CBA [163] selects the bitrate based on the current capacity (*i.e.,* download time of recently fetched chunks). Figure 5.7 measures the difference of the average bitrate between MPTCP and SPTCP. For about 55% of all measurements, MPTCP provides no improvement of the video quality, because either the whole video is already being played at the highest bitrate, or more often (64%), the additional throughput provided by MPTCP is not enough for switching to a higher bitrate. Nevertheless, we do observe less frequent rebuffering events when MPTCP is used, compared to SPTCP: the average number of rebuffering is 0.24, 0.29, 0.21, and 0.18 for SPTCP

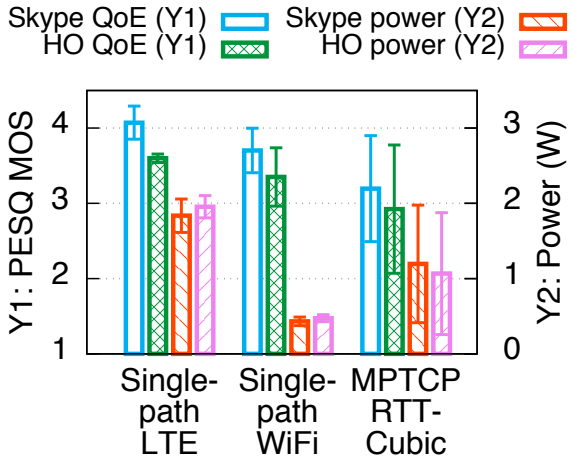


Figure 5.8: QoE and power consumption of VoIP under different SPTCP/MPTCP settings.

over WiFi, SPTCP over cellular, WiFi-primary MPTCP, and cellular-primary MPTCP, respectively. This implies MPTCP can provide more robustness for video streaming.

5.3.3 Other Applications over MPTCP: Voice-over-IP and Instant Messengers

We now study two other important applications (voice-over-IP and instant messengers) in controlled experiments due to the difficulty of capturing their QoE in the user study.

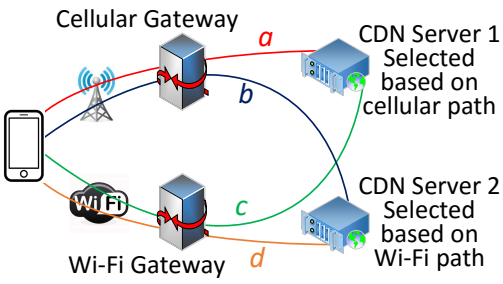
Voice-over-IP (VoIP) is a representative real-time application requiring low latency and small jitter (variation of latency). We study two popular VoIP apps: Skype and Google Hangouts using our campus WiFi and a commercial LTE carrier. The WiFi has smaller RTT than LTE (20ms vs. 60ms) between two clients in our lab: an Ethernet-connected desktop and a Google Nexus 5 smartphone. For both applications, we play a 2-min pre-recorded audio from the desktop client for 10 times and record the received audio at the phone side to compute the QoE using PESQ MOS [36]. We block UDP to force the app to use TCP, and will consider UDP shortly.

Figure 5.8 (Y1 Axis) plots the VoIP QoE using SPTCP and WiFi-primary MPTCP with different schedulers (RTT for minimum-RTT and RR for round robin). We found that compared to SPTCP, MPTCP with the min-RTT scheduler actually worsens the VoIP

QoE. This is due to the increased latency variation, from 24ms for LTE and 12ms for WiFi to 97ms for MPTCP, and additional receiver-side buffering delay of out-of-order packets, from 8ms for SPTCP to 82ms for MPTCP. Both factors contribute to the significant increase of *application-observed* jitter that lowers the PESQ score. We also changed the WiFi and LTE latency, and observed qualitatively similar results. Overall, our findings suggest that real-time applications requiring low jitter may not benefit from MPTCP, in particular when the two paths have diverse latency. To validate this under UDP, we write a custom program that sends a 100-byte UDP datagram every 10ms over single path or multipath with round robin, and measure the jitter and packet out-of-order at the receiver side. We observed when multipath is used, the variation of UDP one-way delay increases by up to 50% compared to SPTCP, and about 50% of the UDP datagrams are delivered out-of-order. Note although UDP itself does not buffer out-of-order datagrams, the application may either buffer or drop them, both resulting in degraded QoE.

Meanwhile, we employ the model used in §5.3.1 to compute the radio energy consumption for each scheme in Figure 5.8. As shown (Y2 Axis), compared to SPTCP over WiFi, MPTCP incurs additional radio power of 125% to 177% for VoIP, making it even more undesirable to use MPTCP.

Instance Messenger (IM). We study three popular IM apps: Facebook, Google Hangout, and Whatsapp. We test them by sending a message every 30 seconds from one phone to another over SPTCP (WiFi only) and WiFi-primary MPTCP with the min-RTT scheduler. We measure the message delivery delay, the key QoE metric, at sender. We also compute the radio energy consumption. We found MPTCP increases the message delivery delay by 5% to 36% compared to SPTCP. This is due to the transmission of small data on the higher-latency LTE subflow (explained in §5.3.1). MPTCP incurs additional radio energy consumption of 18% to 72% for the message delivery. Overall, we found using MPTCP for IM incurs both performance and energy penalty, and thus should be avoided.



Path	*a*	*b*	*c*	*d*
BW (Mbps)	6.0	6.0	3.0	3.0
RTT (ms)	80	80 + *x*	60 + *x*	60

Figure 5.9: Example: CDN sever selection over multipath.

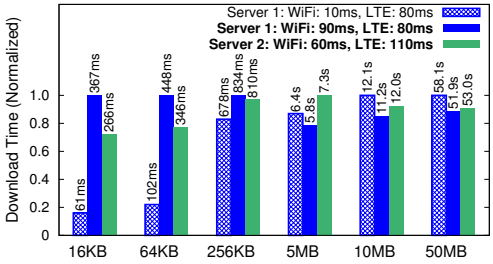


Figure 5.10: File download time from different CDN servers, averaged over 10 runs. Shaded blue is a "what-if" scenario.

Middlebox Friendliness. We also investigate whether the TCP option used by MPTCP can pass middleboxes that are prevalent in cellular networks [184, 94]. We find that both AT&T and T-Mobile middleboxes strip TCP options from TCP header. This motivates multipath solutions that do not require any TCP/IP extensions.

5.3.4 Interplay between Multipath and CDN

In this section, we consider a classic problem of server selection in the context of multipath. Due to the wide use of Content Delivery Network (CDN), the same content (*e.g.,* a web page or a video chunk) is often replicated across servers at multiple geographically distributed locations. A user request will be directed to a CDN location, which is usually close to the client, that provides the best user experience. Most CDNs achieve this by leveraging local DNS (LDNS) server with CDN server selection algorithms based on

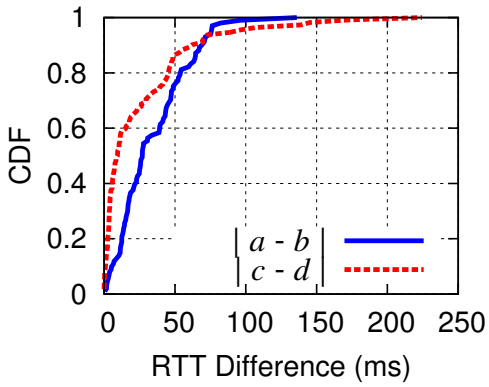


Figure 5.11: Distributions of path latency differences for 190 out of Alexa top-500 websites. Refer to Figure 5.9 for the four paths.

information such as geo-location databases[50].

Measurement for CDN over Multipath. DNS-based server selection can still work in multipath. But here we have a new issue that has not been explored by the literature to our knowledge. Since the DNS request is sent by only one path's (usually the default path's) LDNS server, the CDN is not aware of the existence of other paths. Therefore, although the selected server provides good performance for one path, it may be sub-optimal for other paths and therefore for MPTCP. We conducted an emulated experiment to illustrate this. In Figure 5.9, let Server 1 and 2 be the optimal CDN servers selected based on the cellular and the WiFi path, respectively. If we use Server 1 (Server 2), then MPTCP will send traffic over path a and c (b and d) for cellular and WiFi, respectively. Figure 5.9 also lists the characteristics of the four paths. We assume that a and b (also c and d) have the same bandwidth since they share the same "last-mile" that is usually the bandwidth bottleneck. But when WiFi (cellular) accesses the server selected by cellular (WiFi), a latency penalty x=30ms caused by additional queuing and propagation delay will be added (on path b and c).

We use MPTCP with the min-RTT scheduler to download files with various sizes from both servers. MPTCP uses path a and c for Server 1, and b and d for Server 2. We discuss two scenarios: downloading small files and large files.

Download small files. As shown on the left side of Figure 5.10, when the files are small, CDN Server 2 gives shorter download time than Server 1 does. This is because small files' download time is largely determined by latency. Path d offers the smallest latency (60ms) that helps reduce the file download time. To validate this, we reduce path c's RTT from 90ms to 10ms. Then Server 1 immediately outperforms Server 2 as indicated by the shaded blue bar.

Download large files. For large file download, usually both paths are saturated so the aggregated bandwidth is more important than the latency. In our case, although both servers have the same aggregated bandwidth ($a + c = b + d$), Server 1 still slightly outperforms Server 2, as illustrated on the right side of Figure 5.10. This is explained as follows. The aggregated bandwidth is dominated by LTE. The small LTE RTT offered by path a (compared to path b) shortens the TCP congestion control loop. This helps LTE recover from (real or more often, spurious [96]) losses more quickly, eventually leading to better LTE bandwidth utilization.

The right side of Figure 5.10 also illustrates an interesting phenomenon when we reduce path c's RTT from 90ms to 10ms for large file download. Doing so actually increases the overall download time non-trivially. In other words, improving a subflow in MPTCP may worsen the overall performance! We found this is attributed to the very design of MPTCP's RTT-aware scheduler. The very small RTT on a low-bandwidth path (in our case, WiFi) causes MPTCP to distribute more data onto that path. This can happen, for example, at the very beginning of a file transfer when both paths have empty spaces in their cwnd. As a result, less data is transferred over the high-bandwidth (LTE) path, leading to longer file transfer time.

The above emulation implies that choosing different CDN servers can lead to different network performance, depending on (1) MPTCP's scheduling algorithm, (2) traffic patterns, and most importantly, (3) characteristics of diverse paths. Within them, (1) and (2)

are explained in the above emulation. We conduct another measurement to demonstrate the prevalence of (3), in order to show this is a real-world problem. For each of the Alexa top-500 U.S. websites, our phone sends DNS requests for its landing page over LTE and WiFi, and obtains the corresponding IP addresses of the web servers as IP_{LTE} and IP_{WiFi}, respectively. We found for 190 (38%) out of the 500 websites, their IP_{LTE} and IP_{WiFi} are different. They essentially correspond to CDN Server 1 and 2 in Figure 5.9, respectively. We then measure the RTT difference of path a and b, as well as the RTT difference of path c and d for each of these 190 sites.

The results are plotted in Figure 5.11. As can be seen, the RTT difference is more than 45ms for 20% of the websites, and the difference can be as high as 136ms and 224ms for cellular and WiFi, respectively. Recall that Figure 5.9 assumes the RTT difference is x=30ms. Note the RTT difference measured in Figure 5.11 is much larger than the RTT variation on the same path. Also note for some of the 190 websites, the servers associated with IP_{LTE} and IP_{WiFi} may be co-located for load-balancing purpose, and it is not easy to identify such websites. Therefore, Figure 5.11 is a conservative estimation of the path diversity.

Recommendations. We propose improvements to make CDN server selection multipath-aware. First, we add *protocol support* allowing the CDN infrastructure to learn that a client has multiple network paths, by, for example, adding a backward-compatible extension to DNS protocol. The second and more important question is how to select the optimal server based on characteristics of all paths. Our previous discussion already sheds light on some high-level ideas. For small transfers, the goal is to minimize the delay. Therefore, among servers that are latency-wise closest to each of the client's IP addresses, the CDN can greedily pick the one with the minimum latency as long as that path has reasonable quality. For large data transfers, the goal is to maximize the overall bandwidth. If selecting different servers result in similar overall bandwidth, a possible strategy is to select server based on weighted sum for all paths' bandwidth (a path's weight is negatively correlated

with its latency). Doing so shortens TCP's control loop and leads to higher throughput. We leave detailed design and implementation of a multipath-aware server selection system as our future work.

5.4 MPFlex: A Flexible Architecture For Mobile Multipath

Our measurements in §5.3 reveal several limitations of MPTCP: its interplay with short-lived flows can be further improved; it is important but currently difficult to incorporate application policies into MPTCP; the MPTCP extension is often blocked by middleboxes; MPTCP apparently does not work with other protocols such as UDP; MPTCP also incurs unexpected interaction with CDN.

We realize that many of these challenges stem from the origin of MPTCP, which was originally developed for improving the performance and robustness for datacenter networks [151]. Datacenters are "closed" ecosystems where latency is small, long-lived flows are common for intra-datacenter traffic, and administrators have full control over all applications and network elements. The mobile ecosystem, however, is drastically different, making naïvely porting MPTCP to mobile devices suboptimal.

5.4.1 The MPFlex Architecture

Motivated by the above, we attempt to address an important research question: *what is a better system architecture for mobile multipath?* To this end, we designed, implemented, and evaluated a flexible software architecture of mobile multipath called MPFlex. As illustrated in Figure 5.12, MPFlex is a proxy-based solution. It is transparent to both client applications and servers, with several prominent features.

• MPFlex employs multiplexing to consolidate multiple (potentially short-lived) connections into two long-lived connections: one over WiFi and the other over cellular. The Multiplexed Connections (MC), shown in Figure 5.12), are persistent and are by default

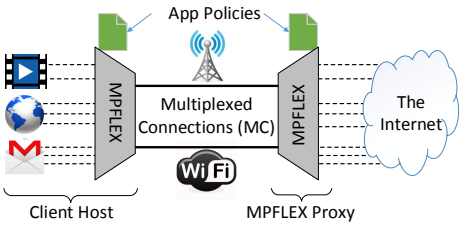


Figure 5.12: The MPFlex architecture.

shared by all applications. They cover the "last-mile" links that are usually the bottleneck. This design overcomes MPTCP's key limitation when dealing with short-lived flows due to following reasons.

First, in MPFlex, since MCs are pre-established, an application connection (shown in dashed lines) only needs one handshake over usually the fastest path to inform the proxy to create the corresponding connection with the remote server, instead of handshakes for *every* subflow in MPTCP. This allows all subflows to be usable sooner, thus improving the bandwidth utilization. This handshake can be eliminated by applying the idea of TCP fast open [150], resulting in *zero-RTT handshake over multipath* between the device and the proxy.

The second benefit of multiplexing is, as an MC is persistent and long-lived, it can preserve the congestion window. This avoids the bandwidth probing (*e.g.,* TCP slow start).Note this advantage also exists in single-path, but it is more prominent in multipath by improving *all* subflows. When a long-lived MC has no data to transmit or receive, the radio interface switches to the IDLE state to save energy, while maintaining the TCP state.

Multiplexing has been employed by other application protocols such as SPDY [178], HTTP/2 [58], and QUIC [39]. MPFlex instead performs multiplexing for the transport layer, agnostic of application protocols, thus applicable to non-HTTP protocols. In particular, unlike a SPDY or HTTP/2 proxy that needs to be man-in-the-middle for SSL/TLS sessions (thus breaking the end-to-end security), MPFlex can transparently work with SSL/TLS.

- MPFlex is a flexible framework. Unlike MPTCP, MPFlex is provided as an OS service

Table 5.3: Comparison of three multipath proxy solutions.

Multipath Solution	Multiplexing	Good Short Flow Performance	Protocol Applicability	User or Kernel	Transparent To SSL/TLS	Middlebox-friendly	Application Policies
MPTCP Proxy	No	No	TCP only	Mostly kernel	Yes	No	No
HTTP/2 Proxy + MPTCP	Yes	No	HTTP only	Kernel + user	No	No	No
MPFlex	Yes	Yes	Any protocol	Mostly user	Yes	Yes	Yes

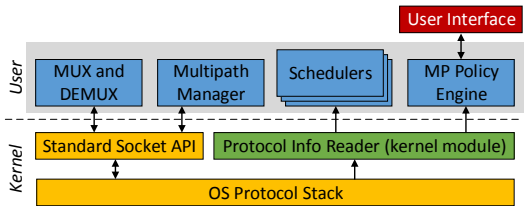


Figure 5.13: Components within an MPFlex endpoint.

and can transparently provide multipath support for non-TCP protocols. An MC can be realized by a wide range of transport protocols such as TCP, reliable UDP, and SCTP [168], enabling continuous transport-layer innovation. Both features are realized through a *pair* of MPFlex modules deployed on both the client and the proxy. They not only perform (de)multiplexing, but also transparently intercept application packets (at the client), and serve as end points of MCs.

- MPFlex has visibility of all client traffic, making it an ideal vantage point for applying user-specified multipath policy based on application usage, performance, cellular data usage, and energy. Realizing such a policy framework by MPTCP itself is difficult unless a centralized traffic manager similar to MPFlex is introduced.

- MPFlex is middlebox-friendly. Since the multiplexing protocol runs *above* the transport layer, it does not require any network-layer or transport-layer extensions such as the MPTCP extension [85] that might not be recognized by today's firewalls and middleboxes. Table 5.3 compares three multipath proxy solutions: MPTCP proxy, MPFlex, and HTTP/2 proxy with MPTCP.

5.4.2 MPFlex Design and Implementation

We implement MPFlex as follows. Multiplexing is performed in a way similar to that in SPDY and HTTP/2, but across TCP connections instead of HTTP transactions (an approach similar to [145]). On the client side, uplink TCP data from applications is segmented and encapsulated into *messages*, which are then distributed onto MCs. Each message has a small header containing its application connection ID, length, and message sequence number. Upon the reception of a message, the proxy performs demultiplexing by extracting the data and forwarding it to the remote server based on the connection ID. Downlink traffic is handled similarly but in the reverse direction. TCP SYN, FIN, and RST are also encapsulated into control messages to realize application connection management.

Based on the above basic multiplexing infrastructure, we developed MPFlex by adding three critical new components shown in Figure 5.13: multipath manager, schedulers, and policy engine. We currently implemented two types of MC: TCP and UDP (for multipath UDP). We only describe TCP here due to space constraints. MPFlex is implemented in C++ with 5000 LoC.

Multipath Manager provides basic multipath support. At client host, MPFlex configures local routing tables, and sets up two regular TCP connections to the MPFlex proxy over the WiFi and cellular interface, respectively, as MCs. An MC is long-lived, unless its network interface is down. In that case multipath multiplexing falls back to single-path. The MC is reestablished when its interface becomes alive. Extending MPFlex for supporting more than two interfaces is also straightforward.

When an application connection issues a TCP SYN handshake, the client-side MPFlex module intercepts it, and sends a control message, containing the server IP and port, to the proxy-side MPFlex module over one MC selected by the scheduler (described below). The proxy then establishes the connection to the remote server. This differs from MPTCP's connection establishment where *every* path needs to perform its own handshake. A sim-

ilar situation happens when closing the connection. The handshake message is treated as transport-layer payload, not bound to a particular path. Therefore, if WiFi is congested while LTE is active but less loaded, the handshake is performed over LTE. Also as mentioned before, MPFlex allows the handshake to be piggybacked with the uplink user data (of up to a threshold of n bytes) to achieve 0-RTT handshake over multipath.

Policy Engine provides a higher-level abstraction of determining when and how to use multipath according to user-defined policies. In our current implementation, a policy is an ordered list of rules specifying what kind of traffic should use which multipath scheme, such as "multipath with minRTT scheduler is only used by browsers and YouTube, and single-path is used in all other cases". For uplink traffic, the client-side MPFlex module can execute the policy by itself. For downlink traffic, instead of letting the proxy perform traffic classification, the client directly instructs the proxy on how to apply multipath by attaching a one-byte label to an uplink message. The proxy then applies the policy to the downlink traffic according to the label.

The policy engine can be extended to consider cellular billing (*e.g.,* disable multipath when the monthly data plan has less than 100MB left), energy (*e.g.,* disable multipath when the battery is low), and performance (*e.g.,* for YouTube, use cellular as the secondary path only when WiFi cannot provide 1Mbps throughput). We plan to realize such metrics in our future work.

5.5 Evaluation of MPFlex

We conduct extensive evaluation of MPFlex. For performance, we compare MPFlex with MPTCP with default settings. All experiments were conducted using real WiFi and LTE networks. We use `tc` to apply bandwidth throttle and to add extra delay on both paths[1] based on recent large-scale measurements of metropolitan LTE [96] and WiFi [164] users.

[1]WiFi: uplink 2020kbps, downlink 7040kbps, RTT 50ms;
LTE: uplink 2286kbps, downlink 9185kbps, RTT 70ms.

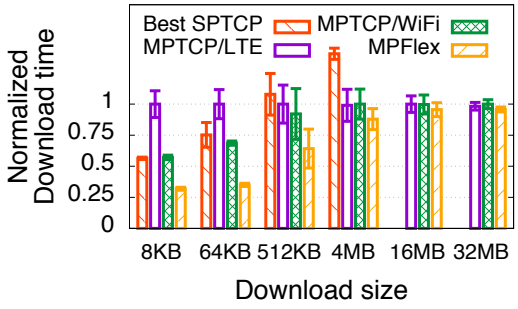


Figure 5.14: Single file download over MPTCP and MPFlex (best SPTCP results shown only for small downloads).

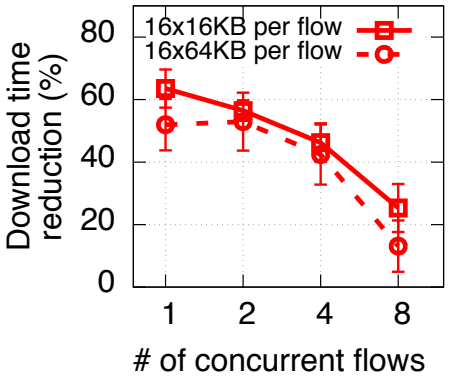


Figure 5.15: Transfer many short flows over MPTCP and MPFlex.

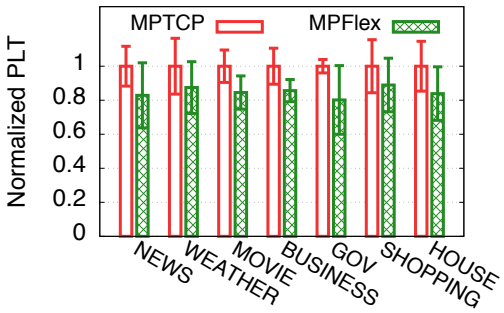


Figure 5.16: Fetch web pages over MPTCP and MPFlex.

The same configurations were used by another recent MPTCP study [92]. For apple-to-apple comparison, MPFlex and MPTCP employ the same min-RTT scheduling algorithm and the same congestion control (decoupled Cubic *i.e.,* each path runs TCP Cubic independently) unless otherwise noted. We next describe the evaluation results.

5.5.1 File Download

Download a single file. Figure 5.14 compares single file download time under three schemes: cellular-primary MPTCP, WiFi-primary MPTCP, and MPFlex, for different file sizes. Compared to the best MPTCP scheme, MPFlex reduces the download time by 11% to 49%, due to its simplified handshake procedure that makes better utilization of both paths, as mentioned early in this section.

Handling multiple short flows. We wrote a custom benchmark tool that generates small flows sequentially or concurrently. Figure 5.15 plots the overall download time for MPTCP and MPFlex under eight traffic patterns. Compared to downloading a single file, when handling *multiple* short flows, the advantage of MPFlex is more phenomenal with download time reduction ranging from 13% to 63%. As mentioned before, this is attributed to two features of brought by MPFlex's strategic multiplexing: simplified handshake (same as the single file download case) and being capable of maintaining the congestion window[2] for multiple connections arriving in a bundle. We also note that the savings reduces a bit when the concurrency becomes higher due to improved bandwidth utilization of concurrent flows.

5.5.2 Web Browsing

How much performance gain can MPFlex offer under realistic applications and traffic patterns? To answer this question, we pick seven diverse websites and load their landing pages using Chrome browser on Nexus 5. To overcome frequent content change and server-side load fluctuation for some sites, we use Google Page Replay to take a snapshot of each site, and host it on our server [15]. To further make our setup realistic, we measure the RTT from proxy (MPTCP or MPFlex) to the real servers, and set the same RTT for the link between the proxy and our server when replaying each website.

[2]Similar to a regular TCP connection, multiplexed connections in MPFlex still conservatively perform slow start after idle period.

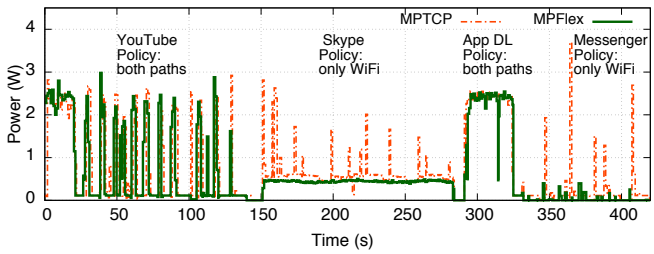


Figure 5.17: Case study: MPTCP applies multipath to all traffic, while MPFlex does that selectively based on user policy.

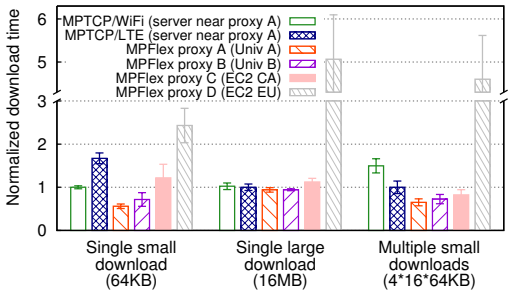


Figure 5.18: Performance impact of MPFlex proxy location.

The results are shown in Figure 5.16. Compared to Figure 5.15, the page load time (PLT) reduction is less, mostly due to the additional browser-side overhead (rendering page, parsing JavaScript *etc.*) and inter-object dependency that often shift the bottleneck from network to local computation [177]. Nevertheless, the improvements are still impressive: compared to MPTCP proxy, MPFlex reduces the PLT by 7% to 20%. We also expect MPFlex will exhibit more advantages on newer mobile devices or tablets where computation is less likely to become the bottleneck.

5.5.3 Applying Multipath Policies

As described in §5.4.2, we have implemented a framework that allows applying different multipath policies at a per-process basis. We demonstrate its effectiveness of saving energy by conducting a case study as follows. We consider four apps: YouTube (playing a 150-second 1080p video), Skype (2-min VoIP call), Google Play (downloading a 50MB app) and Facebook Messenger (sending a message every 30 seconds). We enforce the fol-

lowing policy: use multipath for YouTube and Google Play, and single-path (WiFi) for Skype and Messenger. We compare the radio energy consumption of system-wide MPTCP (applied to all four apps) and MPFlex with the above application-aware multipath policy. As shown in Figure 5.17, MPFlex reduces the radio energy consumption for Skype and Facebook Messenger by 34% and 78%, respectively, while incurring no QoE degradation as explained in §5.3.3.

The policy framework can be extended to consider other factors such as billing and battery life. MPFlex can also enforce the policy at a finer granularity. Consider two use cases. (1) Let Chrome browser to use multipath only for `cnn.com`. (2) Disable multipath for all ad traffic. Both use cases can be realized by controlling multipath usage at a per-HTTP-session basis without modifying the apps. First, for an incoming HTTP session, the client-side MPFlex module obtains its associated domain name. This can be realized by examining directly the HTTP request or the TLS/SSL certificate for HTTPS. Second, it consults a local policy database to determine which multipath scheme to use. Third, it (the MPFlex client) informs the proxy of the policy for this HTTP session using a small label attached to the multiplexed message (§5.4.2). We are currently implementing this feature by adding a MPFlex API exposed to the user space.

5.5.4 Impact of Proxy Location

Our discussion in 5.3.4 indicates the location of a CDN server (in our case here, the MPFlex proxy) may affect the network performance in particular for multipath. To quantify this, we deployed the MPFlex proxy at 4 different locations referred to as A, B, C, and D. Their physical distances from our client smartphone are 3, 420, 3300, 6700 km, respectively. We also measured the minimum WiFi RTT between the mobile client and them to be 27, 44, 81, and 131ms, respectively, and the corresponding cellular RTTs are 49, 66, 100, and 148ms, respectively. We fix the RTT between the proxy and the server to be 4ms, by assuming the server is a nearby CDN node. A recently study measured the

median RTT between a mobile carrier gateway and 30 popular content providers' servers to be ∼4ms [145]. We also evaluate a non-proxy configuration by letting the client directly connect to the server near Proxy A, which has the smallest latency from the client, using the default MPTCP.

We consider three workloads shown at the bottom of Figure 5.18. As shown, for small download file download(s) whose performance is latency-sensitive, the proxy location matters. Nevertheless, despite being further away, Proxies B and C still achieve better or similar performance compared to the default MPTCP configuration, because the benefits of MPFlex outweigh the penalty of additional latency for B and C. Proxy D exhibits low performance because it is located at a different continent. The transoceanic link shifts the bottleneck from the last mile to the Internet.

5.5.5 System Overhead

Despite being realized mostly at user level, MPFlex itself incurs negligible runtime overhead. We monitor CPU usage of a Nexus 5 phone with MPFlex enabled. Compared to the default MPTCP, no noticeable CPU usage increase was observed when downloading large files at high speed (∼30Mbps). The MPFlex protocol overhead, defined as the total message header size divided by the size of all transferred messages, is measured to be less than 1% across 20 Android apps we tested. Multiple instances of MPFlex can be deployed in geographically distributed clouds to achieve scalability.

5.6 Concluding Remarks

We make contributions to mobile multipath research in two aspects: crowd-sourced measurement and software architecture. The user study provides valuable insights of how well multipath works "in the wild". Based on the findings, we introduce new system concepts and research problems such as multiplexed multipath and multipath-aware server selection. We are working on a full-fledged multipath policy system by leveraging the

infrastructural support of MPFlex, and plan to deploy it on the user trial to study how to derive good policies to improve application QoE over multipath, while minimizing the energy overhead.

CHAPTER VI

Design and Implementation of an HTTP-based Multipath Solution for Mobile Devices

6.1 Introduction

According to Cisco Visual Network Index [7], in 2017, 77% of all consumer IP traffic has been video and by 2021, this number will reach to 82%. By 2021, traffic from wireless and mobile devices will also account for more than 63% of total traffic. Given this high demand for online video streaming, video content providers try to satisfy their users by enhancing the quality of their video content (*e.g.,* 4K) and delivering the video content in new forms (*e.g.,* 360°, AR/VR). However, on the other end, the service provided by ISPs and carriers does not seem to keep up with the technology and users' demand.

As an example, to watch 1080p HD and 4K HD videos on Hulu, the *minimum* bandwidth required are 6Mbps and 13Mbps, respectively [18]. However according to an article [1], in which a comprehensive list of chains (*e.g.,* fast foods, coffee shops, store retailers, etc.) that provide the best free WiFi access are ranked by their download speed, only 3 out of 16 chains can provide higher bandwidth than 4.78Mbps, which is far from satisfactory.

Since multiple access technologies (*e.g.,* WiFi, LTE) are widely supported by today's mobile devices (*e.g.,* smartphones and wearables), Multipath TCP (MPTCP) can be used to increase the overall bandwidth. MPTCP has shown to be an effective solution to improve

throughput of the mobile users [70, 77]. However, since its standardization by IETF in 2013, its deployment has been rather slow, mainly due to the fact that it requires both server side and client side modification at the kernel level. In fact, to make any change to the protocol, such as supporting other protocols (*e.g.,* UDP or QUIC [76]) or services (*e.g.,* Anycast [80]), it requires kernel modification. Additionally, MPTCP implementation in Linux and Android provides a coarse-grained API, which limits applications' ability to enforce their policies. For instance, applications cannot limit the data on each interface or more importantly, enable or disable MPTCP for each connection.

Currently, MPTCP scheduler resides in the kernel space and for downlink, the scheduler on the server side decides on which subflow to send data. We argue that to provide more flexibility for the applications to enforce their policy and to make the solution easier to adopt, the scheduling logic should be moved from transport layer to application layer and from the server side to the client side.

In this chapter, we propose MP-HTTP, a novel HTTP-based multipath scheduler. MP-HTTP is easy-to-deploy. It is implemented as an HTTP client and can be included into any app. It is compatible with recent version of Android (>=5.0) and all commercial web servers supporting HTTP/2.0. It does not require any server side or client side modification to the operating system or web server. MP-HTTP is middle-box friendly, as it operates on top of HTTP.

In MP-HTTP, contrary to MPTCP, the connections established over WiFi and cellular are completely decoupled. Additionally, the scheduler's requests over WiFi and cellular may be directed to and served by two different servers, as the best server to reach over each interface may differ. This addresses the MPTCP sub-optimal server selection problem that was reported before [132].

MP-HTTP relies on HTTP byte range request to simultaneously fetch data chunks over WiFi and cellular interfaces. To achieve simultaneous chunk completion time on WiFi and cellular interfaces and fully utilize their bandwidth, MP-HTTP strategically fetches data

chunks on WiFi and cellular interfaces through multiple byte range requests. To account for changing network condition on WiFi and cellular interfaces, MP-HTTP continuously "offloads" parts of the data from the slower interface to the faster interface and dynamically re-adjusts the size of the data chunk that is being downloaded on the slower interface. To minimize the delay between the download of consecutive data chunks on an interface, MP-HTTP leverages various features provided by HTTP/2.0, including multiplexing, PING frames, and flow control.

Specifically, we made the following contributions in this chapter:

- We design a client side multipath scheduler on top of HTTP/2.0 that aims to achieve simultaneous chunk completion time on WiFi and cellular interfaces and fully utilize their bandwidth.

- We implement MP-HTTP scheduler for Android devices and compare its performance with MPTCP and other HTTP-based schedulers by conducting extensive measurements in realistic settings.

- As one of the main applications of MP-HTTP, we further optimize its performance for ABR video streaming. We include MP-HTTP into a popular video player framework and show that under realistic bandwidth settings, its performance is close to MPTCP and it outperforms other state-of-the-art HTTP-based schedulers by up to 44%.

6.2 Motivation

We begin by presenting arguments, backed by measurement results that motivates the need for designing a client side scheduler on top of HTTP, which works with protocols such as UDP, supports multi-homing, and is easy to adopt.

6.2.1 MPTCP Adoption

After five years since MPTCP was standardized by IETF and despite its potential for improving the performance of mobile devices and data center networks, it adoption has been very slow. In 2015, Mehani *et al.* [121] scanned the Alexa top 1M websites and reported that less than 0.1% of the hosts on the Alexa list currently support MPTCP. Recently, we repeated the same measurement on Alexa top 500 websites and observed that this number has not changed much since 2015. Only one out of 500 websites (0.2%) support MPTCP.

Additionally, MPTCP is not a middlebox-friendly solution. As reported by [132], major commercial cellular carriers in the US strip TCP options, which is used by MPTCP, from the web traffic. Thus, establishing MPTCP subflows on the cellular interface to the default HTTP and HTTPS ports are not possible.

Further, MPTCP requires modifications in the kernel stack of both client and server side. It lacks a flexible interface for the apps to enforce their policies [132], as all the implementation and logic reside in the kernel space. All these shortcomings and limitations may explain why the MPTCP adoption has been slow and motivate us to design an easy to adopt solution which does not require any modification in the server and client side.

6.2.2 CDN Server Selection

Here, we study another aspect of MPTCP performance. We investigate the performance of Content Delivery Network (CDN) server selection in the context of multipath.

To accelerate the delivery of web contents (*e.g.,* a web page or a video chunk) with global reach to the users, the content is replicated across servers at multiple geographically distributed locations. Then, the traffic of a user trying to fetch that content will be directed to a server which is usually close to the user and provides the best user experience. Most CDNs rely on DNS to select the best server for a user request and balance the load on their servers.

DNS-based server selection can still work in multipath, but as shown by [132], it may

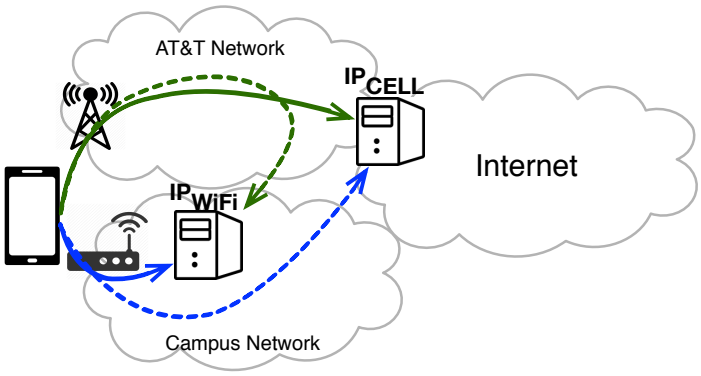
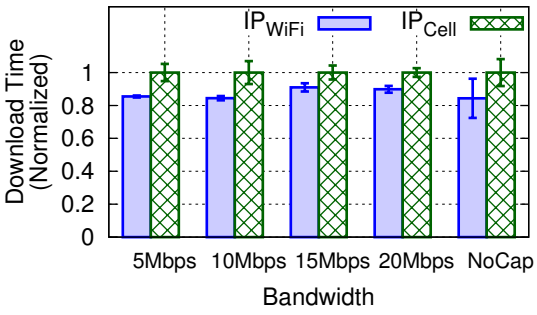
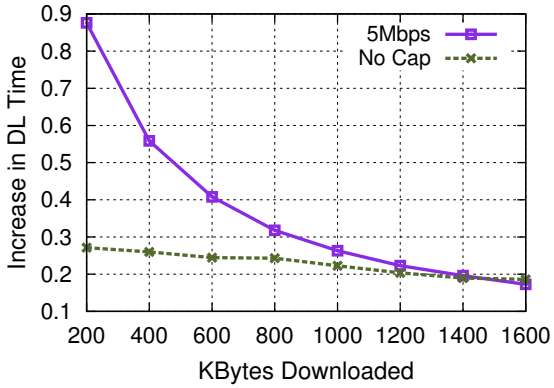


Figure 6.1: Netflix server selection in the context of MPTCP.



(a) Downloading a 1.6MB video chunk from IP_{WiFi} and IP_{Cell} over WiFi



(b) Increase in the download time of data with smaller sizes as we download a 1.6MB video chunk from IP_{Cell}.

Figure 6.2: Impact of sub-optimal server selection on the download time of Netflix video data chunks.

not be optimal. The problem is that when trying to fetch a URL using MPTCP, the DNS request to resolve the domain is only sent over the primary path and is resolved by the primary interface LDNS server. Thus, the CDN's authoritative DNS server is not aware of the existence of other paths and can only consider the IP address of the primary interface LDNS server to find a nearby content server. Therefore, although the selected server provides good performance for one path, it may be sub-optimal for other paths and therefore for MPTCP.

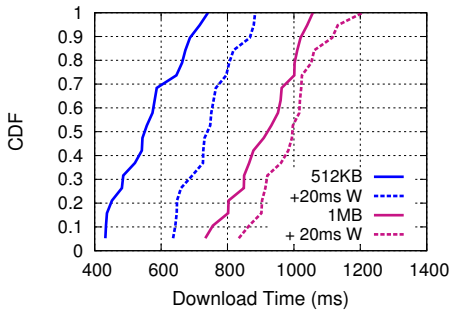


Figure 6.3: Impact of sub-optimal server selection on the performance of MPTCP (emulation).

6.2.2.1 Case Study: Netflix

Putting the content close to the end-users, *i.e., edge*, is the main techniques that CDNs use to accelerate the delivery of web content and reduce the latency between end-users and content servers. To achieve that, CDNs deploy their servers in strategic locations. For instance, Netflix, which accounts for more than 37% of all downstream internet traffic in North American [42] and has its own CDN (*i.e.,* Open Connect [31]), partners with ISPs and deploys its content delivery infrastructures inside the ISP's access network (Embedded Open Connect Appliances), or peers directly with ISPs at one of their peering exchange points (IXP-based peering).

When these content servers are located inside an ISP, the problem of multipath server selection becomes worse, as the traffic of other subflows will be detoured into the primary interface access network. As a result of that, the latency of other subflows to the content server will increase.

Figure 6.1 shows an example of Netflix deployment for an ISP and cellular carrier. Here, a smartphone user is connected to a campus network through WiFi and to AT&T network through cellular. To find where these content servers are located, we use the information included in the reverse domain names of the IP that video chunks are fetched from [59]. As shown in this figure, over WiFi, users' traffic is directed to a content server inside the campus network (*i.e.,* IP_{WiFi}) and over cellular, their traffic is directed to a con-

tent server in an internet exchange point (*i.e.,* IP_{CELL}). In this scenario, round-trip latency of users to IP_{WiFi} is around 4ms over WiFi and around 40ms over cellular. Similarly, round-trip latency of users to IP_{CELL} is around 25ms over WiFi and around 40ms over cellular. Now, assume that both Netflix content servers and users' smartphones are equipped with MPTCP capability and cellular is the primary interface. In this case, users' requests over both WiFi and cellular will be served by IP_{CELL}, which results in 9x increase in the latency of the WiFi path.

To understand how much this extra latency affects the download time of video chunks, we fetch a 1.6MB data chunk [1] from the IP_{CELL} and IP_{WiFi}. Figure 6.2 compares the download time of fetching the file from these two servers over WiFi and under different bandwidth values[2]. As shown in Figure 6.2a, the extra $\sim$20ms latency from the client to the IP_{CELL} increases the download time from 10% to 18%. Figure 6.2b shows the increase in the download time of chunks with smaller size, as we download the 1.6MB data chunk, can be up to 87% for the first 100KB.

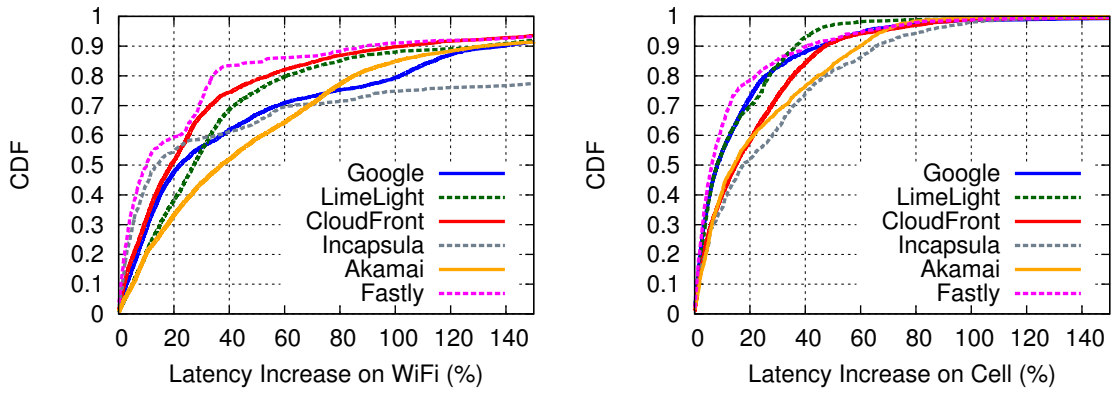
We argue that a better server selection scheme would be to let each interface do the DNS look up separately and fetch the content from the server selected by the its DNS resolver. In that case, the request over WiFi will be served by IP_{WiFi}, which is much closer to the user, in terms of end-to-end latency $\sim$4ms vs. $\sim$25ms).

To compare these two cases and show how much MPTCP performance (*i.e.,* download time) will be affected by the sub-optimal mapping, we did an emulation. We setup an Apache web server with MPTCP support inside the campus network, which is identical to IP_{WiFi} in terms of round-trip latency. To emulate IP_{CELL}, we use the same server, but we add extra 20ms latency to the WiFi traffic using `tc`. By adding this 20ms latency, we emulate the round-trip latency of users to IP_{CELL} over WiFi.

Figure 6.3 shows the impact of the 20ms extra latency, caused by selecting IP_{CELL}

[1] We could only find a limited number of video chunks' URLs hosted by Netflix servers, as the URL of video chunks are usually signed with various client and server side properties and have an expiration time and cannot be requested from different networks and devices [2].

[2] We cap the WiFi bandwidth on the WiFi router using `tc`



(a) Latency increase of IP_{Cell} compared to IP_{WiFi} over WiFi.

(b) Latency increase of IP_{WiFi} compared to IP_{Cell} over cellular.

Figure 6.4: Crowd-sourced latency measurement of multipath server selection.

Table 6.1: Results of DNS lookup over WiFi and cellular for each CDN.

CDN	**Different IP over WiFi and cell**	**Different IP within the same /24 prefix**
Google	87%	5.4%
LimeLight	35%	0.4%
CloudFront	77%	0.0%
Incapsula	21%	0.0%
Akamai	88%	6.4%
Fastly	47%	0.0%

over IP_{WiFi} for the WiFi subflow, on the download time of a 512KB and a 1MB file using MPTCP. As shown, for the 512KB and 1MB file, mean download time is increased by 31% and 9%, respectively. Here the impact of the sub-optimal server selection on the download time of the smaller file is higher. This is caused by the fact that we are using a single server for this emulation. In practice, as shown by previous work [135, 88, 62, 171], wide area network latency or distance between the client and server can also affect the throughput and download time of the large files (*e.g.,* video chunks). Additionally, with the recent advances in the access network technology, *e.g.,* millimeter wave and 5G, the performance bottleneck is shifting from the last mile to the wide area network.

6.2.2.2 Crowd-sourced Measurement Study

To understand how prevalent this sub-optimal server selection would be across different CDNs and network providers, we did a crowd-sourced measurement study using an app we released on Google Play. Since MPTCP is not adopted by these CDNs and finding files with sufficient size on these CDNs is not easy, our measurement is limited to measuring latency.

First, for each popular CDN provider, we find at-least one domain that is hosted by the CDN. Then, we schedule the following measurement task, which takes the domain name as input, on the Android devices running our app. For each measurement task, the device first does a DNS lookup of the domain over WiFi and cellular and obtain the corresponding IP addresses of the domain as IP_{WiFi} and IP_{CELL}, respectively. If the domain is resolved to different IP addresses over WiFi and cellular, the client will then measure the RTT difference of the path toward IP_{WiFi} and IP_{CELL} over cellular. It repeats the same latency measurement over WiFi.

We collected the data for more than 5 months from September 2017. In total, 123K measurements are collected from 138 devices, covering 71 carriers across the world. In this study, we covered six popular CDN providers: Google, LimeLight, Amazon CloudFront, Incapsula, Akamai, and Fastly. We find for 48% of measurements, the domains resolved to different IPs and out of the 48%, only 3% are within the same /24 prefix. Table 6.1 shows the statistic for each CDN. As shown in the table, for different CDNs, the percentage of DNS lookups that resolved to different IPs ranges from 21% for Incapsula to 88% for Akamai. This is attributed to the fact that different CDNs have different number of points of presence (PoPs) across the world. Surprisingly, only a small number of these different IPs belong to the same /24 prefix.

Figure 6.4 plots the distributions of the increase in the round-trip latency of the secondary path when its traffic is directed to the server that is selected by the primary path's LDNS. As can be seen, over WiFi (Figure 6.4a), median increase is between 10% to 37%.

Median increase for cellular (Figure 6.4b) is less than that of WiFi (6% to 17%), as round-trip time over cellular is usually longer. Both of the Figures also exhibit a long tail.

6.2.3 Anycast and Load Balancing

MPTCP is not compatible with load balancers and any cast servers. In Anycast, a single IP address is assigned to multiple hosts in different locations and this IP address is announced through BGP. In this case, when using MPTCP, the handshake traffic of primary and secondary subflows may be routed to different physical servers, that advertise the same IP address. Since the server that is reached by the secondary subflow is not aware of the MPTCP connection and is not maintaining the connection state, the secondary subflow will not be established. In the case of load balancing, we have a similar situation, where the subflows with different 5-tuple, *i.e.,* protocol, source and destination address, source and destination port, may be assigned to different servers. This problem is recently addressed by [80], however the solution requires changes in the anycast's and load balancers' implementation, as well as modification in the Linux kernel, which makes it difficult to be adopted.

6.2.4 Flexible Transport Protocol Support

Adding multipath capabilities to various application and transport layer protocols has shown to be an effective way to improve their performance and reliability. For instance, Multipath QUIC (MPQUIC) [76], which is an extension to the QUIC protocol and is introduced recently, allows hosts to exchange data over multiple interfaces or networks over a single connection. It has shown that MPQUIC outperforms MPTCP in most situations (*i.e.,* for small file downloads or under high packet loss) [76].

One of the main limitations of these multipath solutions is the fact that adopting them requires modifying the server and client side code. For MPTCP, it requires modifying the kernel on both client side and server side and for MPQUIC, it requires modifying the

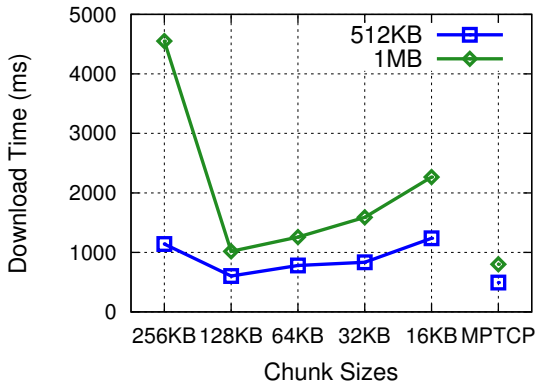


Figure 6.5: Comparing MPTCP performance with constant chunk size scheme with different sizes when downloading 512KB and 1MB files.

server side code. We argue that implementing the multipath scheduler in the application layer and on top of HTTP has two benefits. First, HTTP is the main transfer protocol for web and is flexible to its underlying transport protocol. Thus, by implementing the scheduler on top of HTTP, it can automatically work with any transport layer protocol, including TCP or QUIC/UDP. Second, incorporating a new HTTP library with multipath support requires minimal change in the client side application source code and it does not require any changes in other layers of the client side and server side code.

6.3 Design

To simultaneously utilize WiFi and cellular interfaces to fetch a file, we can simply split the file into two equal chunks and then start downloading each chunk on each interface using HTTP byte range request. However, with equal chunk sizes, we can only achieve optimal download time if the two interfaces have the same bandwidth and latency. Otherwise, one interface will complete the download of its chunk sooner and so its bandwidth will be under-utilized.

To better account for different path properties and changing network condition, we can further split the file into smaller chunk sizes [109]. With smaller chunk sizes, more chunks will be downloaded on the faster interface, thus we may get closer to achieving

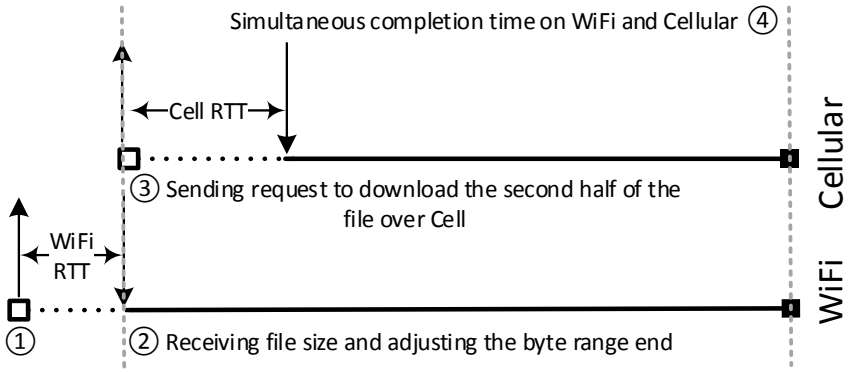


Figure 6.6: Timeline of sending HTTP requests to download data chunks on WiFi and cellular interfaces.

simultaneous chunk completion time at the receiver side. However, this will incur additional overhead of sending more HTTP requests to fetch these data chunks. Sending an HTTP request causes additional delay of at-least one RTT between the download of each consecutive chunks.

Figure 6.5 shows how download time changes with smaller chunk sizes, when a client downloads a 512KB and a 1MB file with 10Mbps and 5Mbps as the bandwidth of cellular and WiFi networks, respectively. As can be seen, for both files, download time initially decreases with the chunk size of 128KB, but with smaller chunk sizes, download time increases, as the overhead of sending more HTTP requests outweighs their benefit of achieving simultaneous chunk completion time on both WiFi and cellular interfaces. Note that optimal chunk size changes over different network conditions and file sizes, thus it is not possible to fine a chunk size that is optimal under any conditions.

To achieve optimal download time, download of the chunks must be completed at the same time on both interfaces [91]. Thus, the number of bytes to download on each interface should be proportional to its bandwidth. More precisely, assuming that we only download one chunk over each interface and considering the delay caused by HTTP request for each chunk, the size of the chunk on interface i must be:

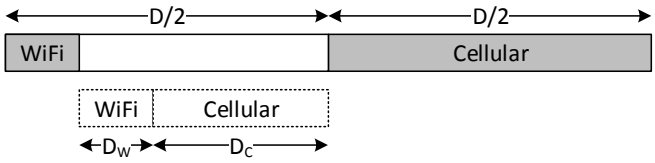


Figure 6.7: Two iterations of the byte range adjustment. Grayed area shows the downloaded parts of the data chunks.

Algorithm 1: Scheduling algorithm

Input: Interface $this$ and $other$ that transmit data over either WiFi or cellular. This function is called when we receive bytes on $this$ interface.

Output: A new stream to be created on the fast interface if $this$ is faster than $other$, otherwise NULL.

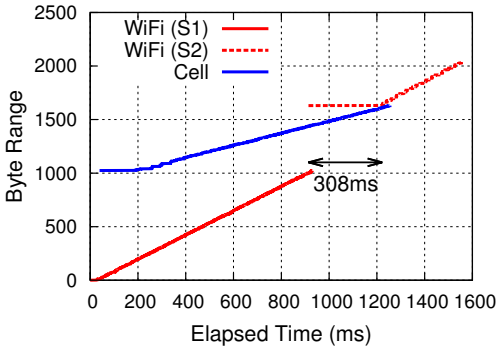
```
1 thisDLTime ← bytesToDL(this) / getBw(this);
2 otherDLTime ← bytesToDL(other) / getBw(other);
3 if otherDLTime − thisDLTime > δ then
4     if thisDLTime <= estimatedReqDelay(this) & numStreams(this) < 2 then
5         newBytesToDLOnThis ← computeBytesRatio(bytesToDL(other),
          getBw(other), getBw(this), estimatedReqDelay(this));
6         adjustByteRangeEnd(other, newBytesToDLOnThis);
7         return createNewStream(this, newBytesToDLOnThis);
8 return NULL;
```

$$D_i = \frac{DBW_i + BW_iBW_j(RTT_j - RTT_i)}{BW_i + BW_j} \tag{6.1}$$

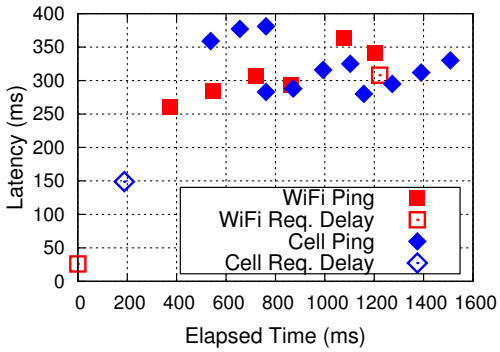
where j is the other interface, $D = D_i + D_j$ is the file size, BW_i, RTT_i, BW_j, and RTT_j are the latency and bandwidth of the interface i and j, respectively.

However, using this equation requires knowing (1) the file size and (2) latency and bandwidth of each interface, ahead of time.

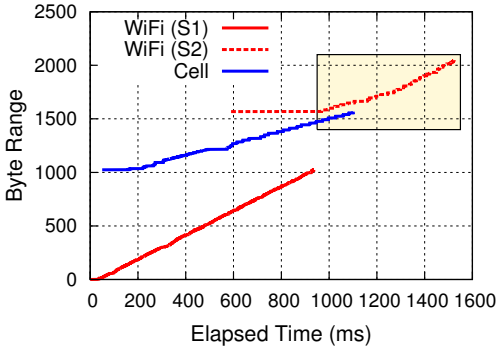
While file size can be learned from HTTP response header in one RTT, getting an accurate estimate of WiFi and cellular bandwidth before starting to download the file is hard, given the highly variable and unpredictable nature of cellular and WiFi bandwidth.



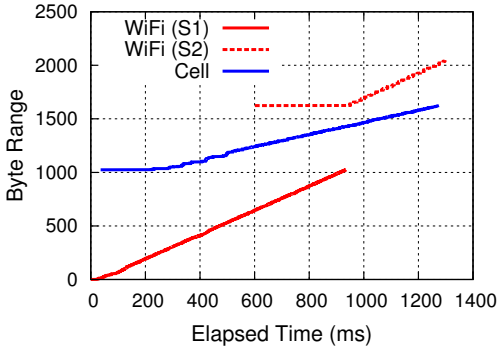
(a) Performance with Incorrect estimation of HTTP response latency.



(b) HTTP Ping measurement.



(c) Performance with HTTP Ping measurements. Highlighted area shows the impact of tail bytes.



(d) Optimal performance with accurate estimation of HTTP response delay and tail byte elimination.

Figure 6.8: Performance of MP-HTTP scheduler for downloading a 2MB file on a device connected to 10Mbps WiFi and 5Mbps cellular network. Y-axis shows the byte ranges requested on different interfaces.

6.3.1 Our Algorithm

To fetch a file, we assume that the file size and bandwidth and latency of WiFi and cellular interfaces are unknown. Thus, we cannot use Equation 6.1 to find the optimal size of the chunks before starting to download the file. Since the file size is unknown, the client has to send an HEAD request to learn the file size and then, as we do not know what proper chunk size is to fetch on each interface, the client splits the file into half and starts downloading each half on separate interfaces. By downloading half of the file on each interface, we achieve two goals. First, we will have enough samples to get an accurate estimation of the bandwidth on each path, and second, compared to having smaller and constant chunk sizes, we have to send smaller number of HTTP GET requests, as a result of downloading smaller number of data chunks.

If the bandwidth of the two interfaces are equal, the download of these two chunks will be completed at the same time, thus achieving optimal download time. However, if WiFi and cellular interfaces have heterogeneous properties, then one of them will finish sooner. In that case, after the fast interface completed the download of its chunk, it will start helping the slow interface by downloading a part of the data that is originally assigned to the slow interface. This new chunk is taken from the end of the chunk that is being downloaded by the slow interface.

Since our goal is to achieve simultaneous chunk completion time by both WiFi and cellular interfaces, we find the size of the chunk that is going to be downloaded by the fast interface from Equation 6.1 with two changes. First, D is the remaining data that is not downloaded by slow interface yet. Second, we do not need to include RTT_j, where j is the slow interface. These is due to the fact that we do not need to send a new HTTP request over the slow interface. We measure RTT_i and RTT_j from the HTTP request that client sent on each interface for downloading the first chunks of the data.

6.3.1.1 Re-adjusting HTTP byte range end

To avoid redundantly downloading the same data over both fast and slow interfaces, we need to *re-adjust the byte range end* of the chunk, that is being downloaded by the slow interface, to a smaller value:

$$byte\_range\_end_{\text{slow}} = new\_byte\_range\_start_{\text{fast}} - 1$$

And $new\_byte\_range\_start_{\text{fast}}$ is computed based on the Equation 6.1. Since re-adjusting the byte range is not supported by HTTP/1.1, we use HTTP/2.0 `RST_STREAM` frame to cancel the stream on the slow interface. The `RST_STREAM` frame allows for immediate termination of a stream. It is sent to request cancellation of a stream or to indicate that an error condition has occurred [58]. The client sends this frame once it reaches the updated $byte\_range\_end_{\text{slow}}$ on the slow interface.

Additionally, to learn the file size, instead of sending an HTTP HEAD request, we can save one RTT by requesting to download the whole file on WiFi interface. Then upon receiving the file size information from the `content-length` header field of HTTP response, the client can re-adjust the current byte range end on the WiFi interface to the first half of the file size and then we start downloading the second half of the data on the cellular interface. The timeline of these requests is depicted in Figure 6.6. As can be seen, the request on the cellular interface cannot be sent until the client receives the file size information on the WiFi interface. In §6.4, we will explain how we can further optimize this.

To account for changing network condition on the WiFi and cellular interfaces, the client repeatedly requests a portion of the data that is being downloaded by the slow interface on the fast interface, when the fast interface completes the download of its chunk. Figure 6.7 shows two iterations of this process. As can be seen, initially, the first and second half of the data are requested on the WiFi and cellular interfaces, respectively, as there is no

bandwidth and latency information available to the client. In this example, we assume that cellular interface is the faster interface, so it completes the download of its chunk sooner than the WiFi interface. Once half of the file is downloaded on the cellular interface, the client computes D_C and D_W according to the Equation 6.1. Then on the WiFi interface, it re-adjusts its byte range end to D_W and for cellular interface, it starts downloading D_C. Note that since the connections over WiFi and cellular might be highly variable, in terms of bandwidth and latency, our estimation of the bandwidth and latency may not be accurate, so we may need additional iterations and adjustments to make sure these two interfaces complete the download of the data chunks at the same time.

6.3.1.2 Pipelining HTTP requests

For each iteration, the client has to send an HTTP request on the fast interface to fetch the new data chunk, that is offloaded by the slow interface. Sending an HTTP request has an overhead of one RTT. To mitigate this additional delay caused by each HTTP request, we can simply send the HTTP requests earlier, *i.e.,* before downloading the entire chunk on the fast interface, so that in the client side we can immediately start receiving the data on the fast interface when the download of previous chunk is finished.

When using HTTP/1.1, even with persistent connections, sending an HTTP request, while the client is still receiving data on an existing connection to the same server, will result in creating a new TCP connection. Creating a new connection may degrade the throughput, as the client will experience TCP slow start. To avoid that, we use HTTP/2.0 pipelining support, where multiple *streams* are multiplexed over a single TCP connection. Then, to avoid any gap between the two consecutive streams, we send the request for the new chunk *one RTT* before the estimated download time of the current data chunk, that is being downloaded by the fast interface.

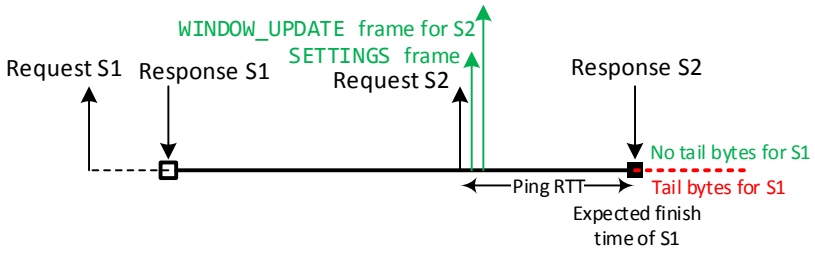


Figure 6.9: Timeline of sending HTTP requests and HTTP/2.0 control frames to download two chunks of data, S1 and S2, on a single connection.

6.3.1.3 Put everything together

We now describe the scheduling algorithm (Algorithm 1). This function is called when the client receives bytes of data on the WiFi or cellular interfaces. Note that at any given time, we have at-most one connection over each interface and each connection may include at-most two HTTP/2.0 streams. Upon receiving data from either WiFi or cellular interface, this function first estimates the completion time of the chunks that are being downloaded over WiFi and cellular interfaces (Line 1 and 2). To estimate that, it uses its estimation of each interface's bandwidth, which is updated as it receives more data on each interface and computed using weighed moving average. Then, it will create a new stream over the current interface, *i.e.,* $this$, if the following conditions hold: (1) the current interface is faster, *i.e.,* it is expected to complete the download of its chunk sooner than the $other$ interface, (2) the difference of the expected finish times is more than a threshold of δ, and (3) the expected time to complete the download of the current chunk is less than the estimated HTTP request delay, so that it can start receiving data from the next chunk as soon the download of the current chunk is finished. If these three conditions hold, it will use Equation 6.1 to compute the amount of data to be downloaded on the fast interface and achieve simultaneous finish time on both interfaces. It also adjusts the byte range of the chunk that is being downloaded by the slow interface so that the same data is not downloaded by both interfaces.

6.3.2 Implementation Challenges

When we implement the proposed scheduler, we noticed that there are two performance issues with the results. First, when the scheduler requests the next chunk one RTT before the expected finish time of a stream, often the client starts receiving data much later than the expected time. Second, when re-adjusting the byte range of the stream over slow interface and then canceling the stream when the scheduler reaches to the updated byte range end, we observe that the transfer of the data does not stop immediately. In fact, the device keeps downloading data for a long time after canceling the stream. We observe that this significantly affects the performance of the next stream. We describe the root causes behind these problems and how we address them, separately.

6.3.2.1 HTTP/2.0 ping measurements to estimate application-layer delay

To predict HTTP response latency (*i.e.,* latency from sending an HTTP request to receiving the response) and how early the request for the next data chunk must be sent in order for the data to be received on time (*i.e.,* right after the finish time of the previous chunk), we use the response delay of the HTTP requests that client initially sends on both WiFi and cellular interfaces to fetch the first chunks. However, for the next consecutive requests over an existing connection, we noticed that the delay increases and it is higher than the expected value. For instance, Figure 6.8a shows download of a 2MB file on a smartphone connected to 10Mbps WiFi and 5Mbps LTE. As can be seen, the HTTP response delay of fetching the first half of the file (S1) on the WiFi interface is 26ms, however, for the next data chunk (S2), it takes 308ms to receive the HTTP response.

This increase in the HTTP response delay, when the client sends a request to create a new stream on a connection that is being used by another stream, is caused by network buffering delay in various places in the network, including TCP sender buffer and buffers within the ISPs and cellular carrier network. Since the initial HTTP request for the first stream does not experience such delay, its delay cannot be used to estimate the delay for

the next consecutive streams.

To account for network buffering delay and be able to estimate that, we use HTTP/2.0 PING frames [58]. The PING frame is a mechanism for measuring round-trip time from the sender, as well as determining whether an idle connection is still functional. In contrary to ICMP requests which operates in the network layer, HTTP/2.0 PING operates in the application layer, thus its frames experience the same delay as the HTTP response delay.

Having an accurate estimate of HTTP response delay can help in finding the proper chunk sizes as well as the right time to send the HTTP request for the next data chunk. Thus, to be able to estimate that, the client sends PING frames periodically. Frequent measurements of HTTP PING frames over both interfaces allows us to get a more accurate estimate of application layer delay. Figure 6.8b shows the HTTP PING samples collected during download of a 2MB file (previously shown in Figure 6.8a). In this Figure, PING frames are sent every 150ms and 100ms on the WiFi and cellular interfaces, respectively. As can be seen, the delay of the second HTTP response is similar to the recent HTTP PING samples collected over the WiFi interface. Note that the overhead of sending these PING frames are negligible.

6.3.2.2 HTTP/2.0 flow control-based tail bytes elimination

In the scheduler design, to adjust the byte range end of a stream, we rely on a stream cancellation mechanism that is provided by HTTP/2.0. To adjusts the byte range end of a stream, once the scheduler reads the bytes from the application buffer and reaches the updated byte range end, it tries to cancel the stream by sending a `RST_STREAM` control frame to the sender. Since a TCP connection in HTTP/2.0 is persistent [58] and it may be shared by multiple streams, when the client cancels a stream, its underlying connection cannot be closed. Thus, when a server application (*e.g.,* Apache web server) receives a `RST_STREAM` frame, it immediately stops sending data of the closed stream. However, all remaining data in the TCP send buffer will still be sent to the client [122]. In fact, the

application server does not have any control over the TCP send buffer, so its data in the TCP buffer cannot be removed. The amount of data that is delivered to the client after canceling the stream, called "tail bytes" by [122], is not limited to the remaining data in the TCP send buffer. It also includes all in-flight bytes being transmitted in the network, which is bounded by the bandwidth-delay products of the network. Note that this problem is specific to HTTP/2.0, as canceling a connection in HTTP/1.1 is equivalent to closing the TCP connection, which results in emptying all its data in the send buffer, leaving much fewer tail bytes to arrive.

Tail byte affects the performance and efficiency of the scheduler in two different ways. First, since the device keeps receiving data of the canceled stream and the TCP connection is shared between the canceled stream and the next stream, the throughput of the second stream will be severely affected. As shown in Figure 6.8c, when comparing the slope of the first stream (S1) with the slope of the second stream (S2), we can see that the throughput of the second stream over WiFi is significantly affected by the first stream's tail bytes. Second, the data that will be received by the client after canceling the first stream is not delivered to the application. Thus this data will be wasted, as it cannot be consumed by the application. Given that cellular data transfer could be expensive, using HTTP/2.0 cancellation mechanism would be very inefficient on mobile devices, financially.

One way to address this problem is to modify the HTTP library so that the tail bytes can be delivered to and consumed by the application. However, it will complicate the design of the scheduler, as it needs to predict how much tail bytes is expected to be received. Instead, we propose using HTTP/2.0 flow control to limit the amount of data being sent to the client.

HTTP/2.0 flow control is a mechanism to prevent the sender from overwhelming the receiver with data it may not want or be able to process [58]. As opposed to TCP flow control, HTTP/2.0 flow control provides *application-level* APIs to regulate the delivery of individual streams, that are multiplexed within a single TCP connection. Also, HTTP/2.0 only provides the building blocks and defers the implementation of a custom flow control to

the client and server. A custom control flow can be used to fetch only a part of a particular resource or to pause and resume the download by reducing and then increasing the stream window size [17].

When an HTTP/2.0 connection is established, the client and server advertise their initial connection and stream-level flow control window in bytes using `SETTINGS_INITIAL_WINDOW_SIZE` frame. Note that flow control can be used for both individual streams and for the whole connection. The window in the server side puts a cap on the amount of data they can send to the client. On the server side, the window size is reduced whenever server sends a `DATA` frame to the client and incremented when the server receives a `WINDOW_UPDATE` frame from the client [58]. When window size becomes zero or negative, the sender cannot send new `DATA` frames until it receives `WINDOW_UPDATE` frames that will cause the window to become positive.

To ensure the application server does not *send* more data than the adjusted chunk size, the idea is to limit its window size, *i.e.,* after the server sends the last bytes in the updated byte range, its window size should become zero. Thus, there will be no bytes outside the adjusted byte range in either TCP send buffer or "in-flight". To set the window size in the server side, we use `WINDOW_UPDATE` frames. However, with `WINDOW_UPDATE` frames, the windows size can be only increased and cannot be decreased. Another option would be to not send any `WINDOW_UPDATE` so that the window size will eventually decreases. However, this would not work, as the initial window size can be set to a very large value (*e.g.,* larger than the file size). To reduce the window size, we use `SETTINGS` frame. `SETTINGS_INITIAL_WINDOW_SIZE` frame alters the window size of all streams sharing a connection. Upon receiving a `SETTINGS` frame, sender updates the size of all windows that it maintains by the difference between the new value and the old value:

$$new\_win\_size = current\_win\_size + new\_initial\_win\_size - old\_initial\_win\_size \tag{6.2}$$

To eliminate the tail bytes, new_win_size must be set to the amount of data that is not downloaded yet on this stream (*i.e.,* $updated\_chunk\_size - bytes\_read$). Here, $current\_win\_size = old\_initial\_win\_size - bytes\_read$. Thus, $new\_initial\_win\_size$ is equal to the $updated\_chunk\_size$.

Therefore, the scheduler sends the `SETTINGS` frame with the $updated\_chunk\_size$ as the new initial window size. To make sure application server receives this frame on time, the scheduler sends it one HTTP PING RTT, that is estimated using the periodic PING measurements, before the expected completion of this stream.

Since `SETTINGS` frame alters the window size of all streams sharing a connection, it will affect the window size and consequently performance of the next stream that is going to be established over the current connection. Therefore, the scheduler has to send a `WINDOW_UPDATE` frame with the default initial window size right after sending the request for fetching the next data chunk. This `WINDOW_UPDATE` frame must contain the stream id of the new stream.

Figure 6.9 shows the timeline of these control and data frame requests to download two chunks of data on a single connection. Note that sending `SETTINGS` and `WINDOW_UPDATE` frames, that are sent alongside with the request for S2, are only required when tail bytes are expected at the end of the S1's transfer, *i.e.,* its byte range end is adjusted to a smaller value.

Figure 6.8d shows the updated results of downloading a 2MB file after implementation of the HTTP/2.0 flow control-based tail bytes elimination. As can be seen, throughput of the second stream S2 is identical to the throughput of the first stream S1.

6.4 Application Specific Optimization and Interaction

In this section, we show how we can further improve the performance of the scheduler using additional information that may be provided by the applications through MP-HTTP API or are collected by the scheduler itself.

One of the identified limitations of MPTCP is the fact that it performs the handshake of the primary and secondary subflows sequantially [132]. It has shown that this leads to low path utilization. Similarly, in our design, since the file size is initially unknown, the scheduler cannot send the requests over the WiFi and cellular interfaces at the same time. As can be seen in Figure 6.6, the request over the cellular interface cannot be sent until the scheduler receives the file size information from the application server. Thus, the sequential requests delays the start of the transfer over cellular by one RTT over the WiFi. This additional delay affects the performance of small transfers more, as latency plays a more important role in the determining the performance of small transfers than bandwidth does.

Having file size information can improve the performance of the MP-HTTP scheduler. As shown in Figure 6.6, when the file size is known, the scheduler can send the requests over WiFi and cellular interfaces at the same time, thus saving one RTT. Additionally, the scheduler does not need to adjust the byte range of the first chunk that is downloaded over the WiFi interface, thus avoiding tail bytes. We show two common scenarios in which the file size information can be provided by the applications.

Another information that might be available to the scheduler is the bandwidth and latency of the WiFi and cellular interfaces. Since an HTTP client can be utilized in different parts of an application source code and for downloading multiple objects, the scheduler may be able to use the collected bandwidth and latency samples for future transfers. The benefit of having an estimate of bandwidth early on is twofold. First, it allows the scheduler to quickly find which interface will complete its download sooner and then sends the request for the next data chunk on-time (*i.e.,* one RTT before the expected completion time of

the data chunk that is being downloaded). Specifically, when the file size is small and PING RTT is high, the scheduler may not have enough time to collect samples for bandwidth and latency estimation. Second, when bandwidth and latency are relatively stable, instead of splitting the file into two equal halves, the scheduler can use Equation 6.1 and find optimal file sizes for each interface. This may result in having smaller number of iterations for adjusting the chunk sizes and achieving smaller download time.

6.4.1 Video Streaming Applications

Since one of the primary reasons to adopt a multipath solution is to increase the available bandwidth and video streaming apps require more bandwidth that other apps, we consider video streaming as the main application of the MP-HTTP. For video steaming apps, video chunks are relatively larger than any other types of mobile application traffic (*e.g.,* web traffic) and with larger file size, all the transport layer (*e.g.,* MPTCP) and application layer (*e.g.,* MP-HTTP) multipath solutions are expected to perform better.

In addition to fetching relatively large files, video streaming traffic have other properties, including sequential download of video chunks over a single persistent connection. As mentioned above, this allows MP-HTTP scheduler to collect more bandwidth and latency samples and use them for the download of future video chunks.

Another property of video players with adaptive bitrate selection support (*e.g.,* DASH) is that the video chunk information are included in a manifest file. Often, manifest file also includes byte range or size of each media segment (*i.e.,* video chunk). Although the standard does not mandate the manifest file to include video chunk size, previous work [187, 183] has shown that segment size is a key component in the adaptation algorithm, as different video chunks that are encoded with the same video quality may have different sizes, thus requiring different bandwidth.

6.4.2 Web based Apps

We consider web-based apps as another application that can benefit from using MP-HTTP. When launching a web-based apps (*e.g.,* news apps), to populate the the screen with the new content, the app initially fetches a manifest file in json format that includes a list of the objects to download [47, 45]. The manifest file may include various properties for each object, including its URL, type, layout and location of the object, size, in case the object is an image, and file size.

As mentioned above, the knowledge of file size improves the performance of MP-HTTP and for web-based apps, size of the file can be provided by the application through MP-HTTP API.

6.5 Implementation

We implement MP-HTTP scheduler for Android devices. To access and transfer data on WiFi and cellular interfaces at the same time, we use an API that is introduced in Android 5.0. Using this API, applications can request to get access to a `Network` object which has a set of `Capabilities` [9]. These capabilities can include different types of transports (*e.g.,* cellular, Ethernet, Bluetooth) and/or properties (*e.g.,* unmetered or not restricted network). Then, if such network with the specified capabilities is available, it will be provided to the application through a callback.

We implement MP-HTTP on top of OkHttp, which is a popular HTTP and HTTP/2.0 client for Android and Java applications [33]. We made modifications to OkHttp source code to expose some of its internal APIs to the scheduler. These includes the functions to read a stream's ID, send HTTP PING, `WINDOW_UPDATE`, and `SETTINGS` frames. Overall, we implement MP-HTTP with about 2,000 lines of Java code.

6.6 Evaluation

We conduct extensive evaluation of MP-HTTP. First, we evaluate the performance of tail bytes elimination, which combines two features of HTTP/2.0: application layer round-trip time measurement using ping frames and flow control. Then, to evaluate the performance of MP-HTTP, we consider two types of workload: single file download and video streaming. For each, we evaluate MP-HTTP under stable network conditions and varying network condition. For stable network conditions, we used `tc` command on the server side to cap uplink bandwidth. Thus, even in the capped bandwidth scenario, the client may still experience last-mile bandwidth fluctuations. For the varying network condition, we used real bandwidth profile traces collected in different locations, including coffee shops, residential apartment, and campus library.

6.6.1 Experimental Setup

We setup our MPTCP testbed using a Nexus 5 phone with Android 4.4.4 and latest available version of MPTCP ported to Android (v0.89.5) and a commodity server with 4-core 3.6GHz CPU, 16GB memory, and 64-bit Ubuntu 16.04 with Linux kernel 4.9.60 that runs a stable version of MPTCP v0.94. To evaluate MP-HTTP, we used another Nexus 5 phone with Android 5.0. Note that MP-HTTP requires Android 5.0 and higher, which is supported by 82% of devices, worldwide [3]. We installed Apache/2.4.29 with HTTP/2.0 support on the server machine. Our implantation is compatible with nginx as well and except enabling HTTP/2.0, our implementation does not require any modification to the web server's source code or any configuration.

6.6.2 Tail Byte Elimination

To examine how the proposed mechanism for eliminating the tail bytes performs, we consider canceling a stream in the middle of downloading a 2MB file. In other words, we intend to change the byte range end to 1MB after we request to download the whole

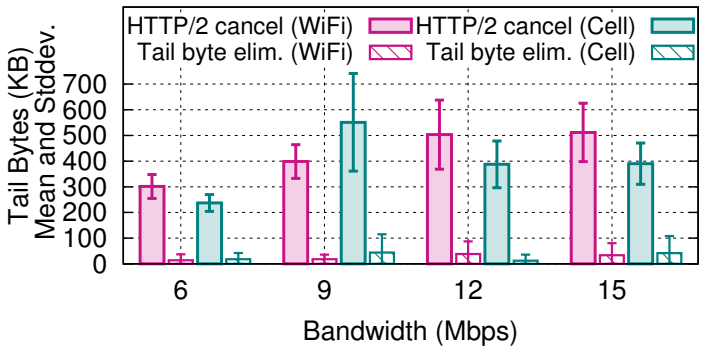


Figure 6.10: Comparing the performance of the proposed tail byte elimination with canceling a stream using HTTP/2.0 `RST_STREAM` frame.

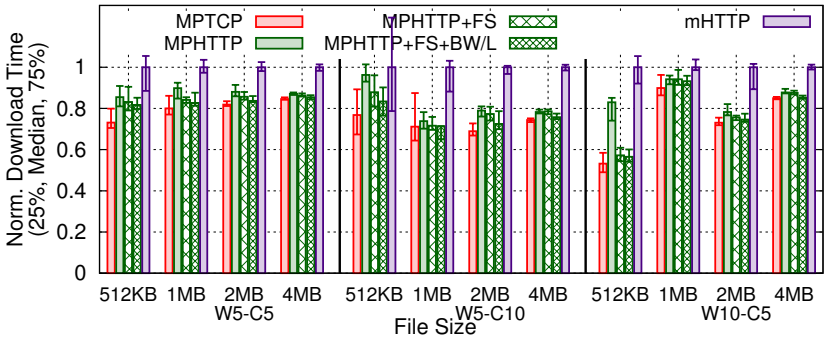


Figure 6.11: Compare performance of different schedulers under three bandwidth settings.

2MB. As explained before, we can send the settings frame with the new file size anytime after establishing the stream. However, the MP-HTTP scheduler has to wait until one RTT before the expected completion time of the current stream to decide the size of the next chunk. Thus, if we can accurately estimate the application layer RTT and completion time of the current chunk and send the `SETTINGS` with the new window size on-time, it is guaranteed that the client will not receive any tail bytes.

Figure 6.10 shows the amount of tail bytes downloaded after canceling the stream using two mechanism: HTTP/2.0 `RST_STREAM` frame and our proposed tail byte elimination. The amount of tail byte is measured when downloading the file over WiFi and cellular and under different bandwidth settings. As shown in the Figure, when using HTTP/2.0 `RST_STREAM`, the amount of tail bytes downloaded over WiFi and cellular ranges from 237KB to 551KB, which translates to 23% to 53% extra downloaded bytes. With the

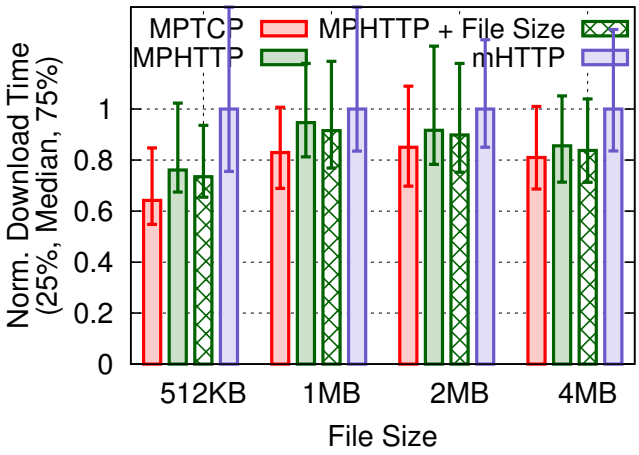


Figure 6.12: Compare performance of different schedulers under real bandwidth profiles.

proposed tail byte elimination mechanism, this reduces to 12KB (1%) to 44KB (4%) extra bytes downloaded.

6.6.3 Single File Download

For single file download, we compare mHTTP [109] and MPTCP with three variants of MP-HTTP, one with the knowledge of file size (*i.e.,* MP-HTTP + FS), one with the knowledge of file size and bandwidth/latency information (*i.e.,* MP-HTTP + FS + BW/L), and the other without the knowledge of file size and bandwidth/delay information. mHTTP uses static chunk sizes and it is shown that with 1024KB as default chunk size, it can achieve maximum performance for downloading different file sizes. However, since we also target smaller file sizes, we try 128KB, 256KB, and 512KB chunk sizes and pick the one with the best performance.

First, we evaluate MP-HTTP under three capped bandwidth settings: 5Mbps WiFi and 5Mbps LTE, 5Mbps WiFi and 10Mbps LTE, and 10Mbps WiFi and 5Mbps LTE. Note that since the main application of MP-HTTP is video streaming, we consider a scenario in which the bandwidth provided by WiFi interface is not enough to stream videos with SD or HD quality [18, 30]. For each bandwidth setting, we download files with three different sizes: 512KB, 1MB, 2MB, and 4MB. For each setting, we repeat the experiment 30 times

and report 25, 50, and 75 percentile of download times.

As shown in Figure 6.11, MP-HTTP can achieve 3% to 26% faster download time than mHTTP. Download times of MP-HTTP are within 2% to 25% of MPTCP's and with larger files, this difference becomes smaller: 7% to 11% for 2MB and 2% to 5% for 4MB. This difference is attributed to the fact that MP-HTTP's time to first byte on each interface is longer than MPTCP, due to two reasons. First, MP-HTTP cannot send the request over the cellular interface, until it receives the file size information from the HTTP response on the WiFi interface. Thus, as shown in Figure 6.6, compared to MPTCP, the transfer over the cellular interface start at-least one RTT_{WiFI} later. Second, contrary to MP-HTTP, MPTCP does not require sending any request on its secondary subflow over the cellular interface. After TCP handshake, MPTCP can immediately receive data on its cellular interface. Therefore, MP-HTTP can start receiving data on the cellular interface $\text{RTT}_{\text{WiFI}} + \text{RTT}_{\text{CELL}}$ after MPTCP does. For smaller files, in which the latency is the dominant factor in determining the download time, the impact of this extra delay on the MP-HTTP performance will be higher.

As discussed in §6.4, we may assume that file size information is available. In that case, MP-HTTP can send the requests over WiFi and cellular at the same time, thus saving one RTT_{WiFI}. Additionally, if bandwidth and latency information are available to the scheduler, either provided as an input from the application or collected from previous downloads, the scheduler can reduce the number of iterations by using Equation 6.1 to find the optimal sizes of the first iteration chunks. As shown in Figure 6.11, with file size and bandwidth/latency information, download time over MP-HTTP is improved by up to 6% in average. Compared to mHTTP, MP-HTTP with file size and bandwidth/latency information can achieve up to 43% faster download time. Overall, MP-HTTP's download time is within 0.1% to 11% of MPTCP's and for 2MB and 4MB files, its download time is within 2% to 5% and 0.3% to 1% of MPTCP's, respectively.

We next evaluate how MP-HTTP performs under five real bandwidth profiles, that are

collected in different locations. The bandwidth at these locations was highly variable, with their standard deviation being up to 57% and 60% of the mean bandwidth for WiFi and cellular, respectively.

As shown in Figure 6.12, when downloading 4 different file sizes, compared to mHTTP, MP-HTTP with and without file size information can bring median download time reduction of up to 26% and 23%, respectively. Compared to MPTCP, the download time of MP-HTTP with and without file size information are within 14% and 18% of MPTCP's for 512KB file and within 3% and 5% of MPTCP's for 4MB file. As can be seen in the Figure, with smaller file sizes, the difference between the download time of MPTCP and MP-HTTP becomes larger. The main reason behind this difference is the delayed request time over the cellular interface, as mentioned before. However, another potential reason is that under highly variable bandwidth, the number of iterations (*i.e.,* re-adjusting the size of data chunk over slow interface and offloading part of the data to the fast interface) increases. For each iteration, MP-HTTP needs at-least one HTTP PING RTT to request for the next data chunk and eliminate the tail bytes. This means that compared to MPTCP, MP-HTTP is slower to react to the changing network condition. This is attributed to the fact that (1) MP-HTTP's scheduler is in the receiver side (*i.e.,* client) for the downlink traffic; (2) MPTCP has a more fine grained control over the data (*i.e.,* packet level control in transport layer vs. byte range level in the application layer).

6.6.4 Video Streaming

Since video streaming is the main application of MP-HTTP, we next compare MP-HTTP's performance with MSPlayer [71] and MPTCP, under server-side capped bandwidth and real bandwidth traces. We use ExoPlayer [12], which provides a pre-built video player for Android with DASH support, to stream a 10min video. According to the manifest file of this video, duration of each video segment is 6sec and it supports 9 different video bitrates, ranging from 253.0Kbps to 10.0Mbps. We replaced ExoPlayer default HTTP client with

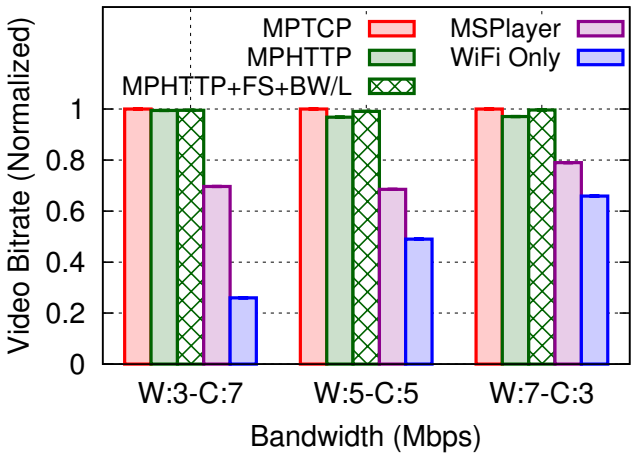


Figure 6.13: Compare the video quality of different schedulers under three capped bandwidth profiles.

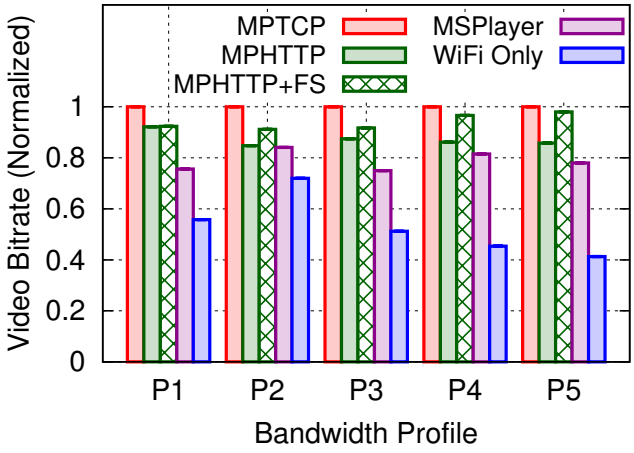


Figure 6.14: Compare the video quality of different schedulers under real bandwidth profiles.

MSPlayer and MP-HTTP by changing a few lines of code. MSPlayer adjusts the chunk size of the fast path based on the throughput ratio. For slow path, the chunk size is doubled or halved based on the comparison between the current bandwidth and its estimated value.

Figure 6.13 shows total video bitrate when streaming a 10 min ABR video under three capped bandwidth settings: 3Mbps WiFi and 7Mbps LTE, 5Mbps WiFi and 5Mbps LTE, and 7Mbps WiFi and 3Mbps LTE. Here, we compare the performance of MP-HTTP with MPTCP, MSPlayer, and single path (*i.e.,* WiFi only). In all the three bandwidth combinations, total available bandwidth is sufficient to stream the highest video bitrate. To compare

how MP-HTTP performs with and without video segment size information, we include the size of each video segment into the manifest file. As shown in the Figure, MP-HTTP always performs better than MSPlayer. Compared to MSPlayer, MP-HTTP with and without segment size and bandwidth and latency information can provide up to 44% and 42% higher video bitrate. This is attributed to the fact that MSPlayer does not pipeline its requests. Thus, its performance suffers from the overhead of sending HTTP requests. The video quality provided by MP-HTTP with and without segment size and bandwidth/latency information are within 0.3% to 0.9% and 0.3% to 3% of MPTCP's.

Figure 6.14 plots the results of comparing MP-HTTP with MSPlayer and MPTCP under five different real bandwidth profiles. Compared to MSPlayer, MP-HTTP with and without segment size information can provide up to 25% and 22% higher video bitrate. MP-HTTP with segment size information can achieve 92% to 98% of the video quality provided by MPTCP.

6.7 Conclusion

In this chapter, we designed a multipath scheduler with the goal of providing an easy-to-adopt solution and achieving similar performance to MPTCP. Toward this goal, we designed and built MP-HTTP, a client-side scheduler that operates on top of HTTP/2.0. MP-HTTP relies on HTTP byte range request to strategically download data chunks on WiFi and cellular interfaces. We demonstrate how we addressed several challenges toward optimal performance while dealing with changing network condition. Through extensive evaluation, we show that MP-HTTP performs close to the MPTCP, in terms of throughput and video quality and outperforms other state-of-the-art HTTP-based schedulers.

CHAPTER VII

QoE Inference and Improvement Without End-Host Control

7.1 Introduction

For network-based applications (apps) like video, Voice over IP (VoIP), and web browsing, knowledge of end users' quality of experience (QoE) is valuable in various ways. When dealing with congestion, any ISP can shape traffic in a manner sensitive to the perceived QoE of its users, *e.g.,* throttling every flow only to the extent that does not significantly degrade QoE for the corresponding users. An app's servers can leverage the knowledge of users' QoE to appropriately adapt its traffic delivery to its users. For example, a video service can reduce the video bitrate to eliminate rebuffering delays incurred at a higher bitrate. Furthermore, if the operating system (OS) on a user's end device can detect when the user is suffering from poor QoE, it can attempt to diagnose the problem.

However, today, all of these useful QoE-aware mechanisms for traffic management, app delivery adaptation, and user experience problem diagnosis are stymied by a basic limitation: determining a user's QoE for a particular app requires software on the user's device that is capable of measuring QoE metrics for that app, and reports information to the entity (OS, ISP, or application server) implementing the QoE-aware mechanism. This limitation stems from several reasons.

- **App-specific QoE metrics.** The metrics that capture user QoE vary significantly across apps, *e.g.,* rebuffering delays for video, mean opinion score for VoIP, and page load times for the Web. This makes it challenging, if not impossible, to write one software, which if installed on a user's device, can measure the user's QoE for any arbitrary app.
- **A lack of API to communicate QoE.** In cases where the user interacts with an app via client software offered by that app's provider, that client is able to measure the user's QoE and relay such information to the app's servers. However, there typically does not exist an interface via which an app's client software can relay measured QoE information to other entities that can make use of this information, such as the user's OS or ISP.
- **Third-party clients.** It can also be challenging for an app's own servers to discover user-perceived QoE because users often access apps via client software not developed by the app provider, *e.g.,* YouTube accessed on Internet Explorer, or a messaging service accessed via a third-party client that has support for several messaging services.

As a result of these limitations, we are currently at an impasse. There is growing recognition that dealing with network traffic based on traditional quality of service (QoS) metrics, (*e.g.,* allocating an equal share of the bottleneck link's bandwidth to all flows, irrespective of which apps those flows correspond to) does not accurately account for users' QoE. Yet, all of the wonderful QoE-aware optimizations detailed above are infeasible to implement today due to the lack of software on end devices which can measure and report QoE to the entity implementing the optimization.

To chart a way forward out of the current impasse, we argue that it is indeed feasible for an entity that has access to a user's network traffic to *infer* the user's QoE, despite not having direct access to app-level QoE measurements from the user's device. Our proposed approach for inferring QoE corresponding to a traffic flow is to rely on models that can map the flow's QoS metrics (such as latency, bandwidth, and loss rate) to the corresponding app's QoE metrics. Here, we define QoE as the objective measures of a specific app's per-

formance that have direct relationships with subjective QoE metrics like user satisfaction and engagement. While a generic QoS-to-QoE model is impractical, our key observation is that such models are indeed feasible on a *per-app* basis.

In this chapter, we first describe how app-specific models that map QoS metrics to corresponding QoE metrics can be generated. Apps often provide multiple features, and each app *usage* may have a different corresponding QoE metric. To find the most common usages from which to build our models, we perform an app usage measurement study, collecting app usage data from 99 real users interacting with 531 apps over 10 days. We then develop *UsageReplayer*, an app-independent tool to replay these user interactions across different apps in testbeds. By varying QoS metrics as we replay user traces, we are able to understand the effect QoS has on the QoE of individual app usages and build our QoS-to-QoE models.

We present results for three types of apps: video conferencing (AppRTC and Skype), on-demand video streaming (playing three state-of-art adaptive streaming schemes), and 55 interactive apps. For video streaming and video conferencing apps, we find significant non-linearities between QoS and QoE, validating the need for our models. For interactive apps, we find that the QoE of these apps are highly sensitive to the changes in end-to-end delay.

Once equipped with per-app QoS-to-QoE models, we present the design and implementation of QOEBOX, a QoE-centric traffic management framework. QOEBOX is a proxy solution and is transparent to both user-facing apps and backend servers. It relies on the QoS-to-QoE models to infer apps' QoE.

Finally, we present how the per-app QoS-to-QoE models, once generated, can be utilized for the purpose of traffic management. We showcase direct applications of the models by implementing two QOEBOX modules: classification and prioritization of apps traffic, and an optimal fair bandwidth allocation scheme. In the former, we show that identifying and prioritizing the traffic of various usages can improve app responsiveness by up to

964%, which translates into 6 seconds of extra delay (§7.5.1). In the latter application, we demonstrate that we can leverage the models to optimally allocate bandwidth to multiple users and improve average video bitrate by 23%, without hurting QoE of any of the users (§7.5.2).

Our contributions can be summarized as follows:

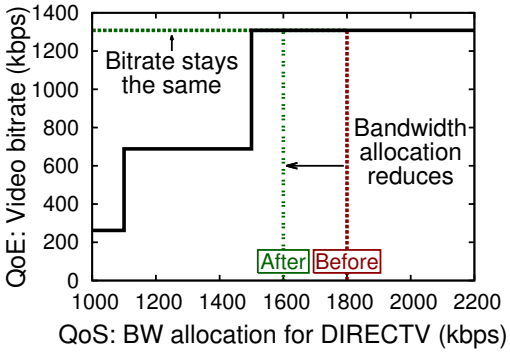
• We propose offline generation of per-app models mapping app-independent QoS metrics to corresponding app-specific QoE metrics. We generate the models by replaying real users' common interactions in popular interactive apps, two popular video conferencing apps, and three video streaming apps, with significantly different QoE metrics. We identify important combination of QoS values that causes change in QoE of these apps using a new adaptive sampling technique.

• We design and implement QoEBox, a proxy-based QoE-centric traffic management framework. QoEBox maps the traffic belonging to an app to its corresponding QoE model, and invokes custom modules to apply traffic shaping policies.

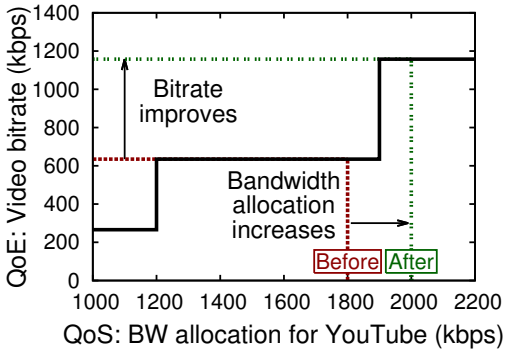
• We showcase how the models can be utilized for the purpose of traffic management by designing, implementing, and evaluating two QoE-aware traffic managements schemes on WiFi access points: prioritizing latency sensitive traffic and an optimal fair bandwidth allocation for bandwidth intensive apps. We show that for both schemes, we can significantly improve various QoE metrics, including frame rate, app responsiveness, and video bitrate, without modifying the end-host or requiring a very precise model.

7.2 Motivating Examples

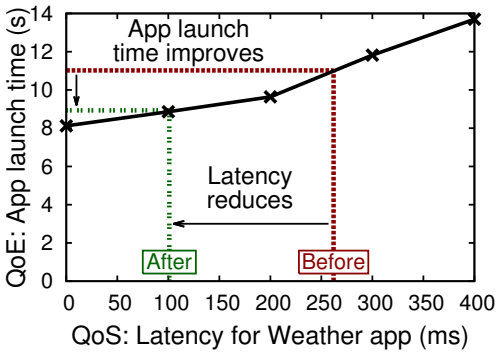
Equipping OSes, ISPs, cellular carriers, and cloud providers with the knowledge of app QoE has several benefits. First, it enables these entities to efficiently allocate resources in a way that maximizes their users' QoE. Second, it provides a feed-back control loop in which these entities can continuously monitor users' QoE, detect and diagnose performance



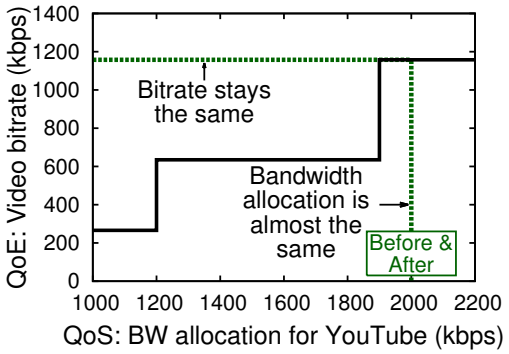
(a) QoE-aware bandwidth allocation for DIRECTV.



(b) QoE-aware bandwidth allocation for YouTube.



(c) QoE-aware traffic prioritization for Weather app.



(d) QoE-aware traffic prioritization for YouTube.

Figure 7.1: Problem of resource management for two scenarios, in which two users using different apps at the same time.

degradation problems, and react by allocating resources appropriately. Third, the knowledge of users' QoE provides an opportunity for application servers to move their adaptation logic from end-user device to cloudlets or base-stations, where they have a more accurate estimate of end-user QoS (*e.g.,* allocated bandwidth).

An important case that can benefit from per-app QoS-to-QoE models is resource management in home WiFi routers. In this section, we use two simple motivating examples of QoE-based traffic management schemes to illustrate the usefulness of QoS-to-QoE models in improving app QoE. In both examples, we consider a scenario where two users are connected to the Internet through a WiFi access point (AP).

Example 1. QoE-aware bandwidth allocation. Suppose two users, user A and B, are watching videos using the YouTube and DIRECTV apps, respectively. The amount of bandwidth each app receives ends up depending on the transport and application protocols. In our example, DIRECTV—which uses Apple HLS—and YouTube both use a single TCP connection to download video chunks, so both get 1800 Kbps from of total link bandwidth of 3600Kbps. Since YouTube and Apple HLS use adaptive bitrate streaming and different encoding schemes, the video bitrates for the YouTube and DIRECTV apps are different—620 Kbps for YouTube and 1277 Kbps for DIRECTV. However, as we see in Figure 7.1(a), the QoS-to-QoE model of Apple HLS indicates that its user can achieve a bitrate of 1277 Kbps with *less* bandwidth. In fact, the extra bandwidth is wasted, in the sense that it is spent on downloading the chunks faster. And as we see from the QoS-to-QoE model for YouTube in Figure 7.1(b), if we were to allocate this extra 200 Kbps bandwidth to YouTube, it can switch to a higher bitrate.

From this example, we see that a change in bandwidth does not necessarily lead to a change in video quality, and with per-app QoS-to-QoE models, an AP can allocate bandwidth in such a way to improve the overall QoE of the system. In §7.3, we show how QoS-to-QoE models are generated, and in §7.5.2 we formulate a bandwidth allocation problem as an optimization problem, which utilizes these models to maximize overall QoE.

Example 2. QoE-aware traffic prioritization. Suppose now that user A is launching the Weather app while user B is watching a YouTube video. The YouTube app aggressively tries to consume all available bandwidth, which results in an increase in queuing delay in the router's buffer. The QoS-to-QoE model of the Weather app shows that its QoE is sensitive to changes in latency (Figure 7.1(c)). Due to extra queuing delay caused by YouTube traffic, the Weather app's launch time increases to 11s. To reduce end-to-end latency of the Weather app, one solution pointed out by previous works [56, 90] is to prioritize its traffic. As depicted in Figure 7.1(d), by prioritizing the Weather app's traffic, we can improve its launch time to 9s, without affecting the QoE of YouTube.

We see from this example that applying traffic prioritization policies can also improve the overall QoE of the system, given per-app QoS-to-QoE models. In §7.5.1, we design and implement a traffic classification and prioritization module that can control the interaction between different types of traffic with different QoS requirements.

7.3 Generating QoS-to-QoE models

In this study, we use *objective metrics* for quantifying QoE, as subjective tests are time consuming and human subjects must be involved in the assessment process. QoE is measured in different ways by different apps: latency and frame rate for video conferencing, video quality for video streaming, and page load time for web browsing. Consequently, a single mapping from QoS to QoE for all apps does not exist. Because of the differences in the protocols used by different apps, even generating separate QoS-to-QoE models for every *app type* is infeasible. For instance, Skype and Google Hangouts use different techniques to deal with packet loss [185]. Even with the same underlying network packet loss rate, users may experience different QoE when using the two apps. Furthermore, apps often provide multiple activities, services, and features to users, each with possibly its own QoS requirement. For example, searching for a video in YouTube vs. playing it, and posting a picture on Facebook vs. updating a status. As a consequence, we focus on generating

QoS-to-QoE models on a per-app basis, and at the *per-usage* level within the app.

We define a *usage* to be a particular interaction with an app, generally with a particular UI component. Scrolling the news feed in Facebook and clicking the play button in YouTube are examples of usages. The set of usages maps to the set of features than an app exposes to users, though some features are not exposed this way (*e.g.,* notifications), and some usages might map to the same feature, depending how the app is designed. We use the expression *usage type* to categorize the usage, with *scroll*, *click*, and *app launch* being examples of usage types.

To generate per-app models, we need to identify the various usages of an app, find the relevant QoE metric for the usage, and understand how various QoS metrics affect that QoE metric. We now describe in detail how we accomplish this.

7.3.1 Recording and Identifying App Usages

There are multiple approaches to identifying app usages. We could rely on app developer support, use an automated state exploration technique, or collect app traces from automated or actual interactive sessions. We chose to collect traces from actual interactive sessions for two reasons. First, collecting traces provides us with the opportunity to replay those traces later, something we rely on when building our QoS-to-QoE models. Second, collecting traces from actual users allows us to understand which apps are commonly used, and which usages are most common. While a monkey testing approach could generate traces for a large number of apps as well, we would not be able to discern the important usages. Understanding which usages are most common allows us to focus on building the models that will provide the largest benefit.

7.3.1.1 Recording App Usage Traces

We record app usage traces using a modified version of the Android framework. We define each touch event as a single *interaction*. To capture interactions, we instrument

the `onTouchEvent` function in the `View` class to record two actions: click and scroll. The `View` class is the parent class of all Android UI components, including widgets—buttons, text fields, etc.—and layouts. So `onTouchEvent` will be called when any of `View`'s child classes is touched. This way, we can capture touch actions with arbitrary apps without instrumenting app source code.

To uniquely identify the touched UI component and replay the action later, we fingerprint and record each UI component that is interacted with. The fingerprint consists of multiple attributes of the UI component. Depending on what is available, it can include: (1) the app name, (2) the `Activity` class name, (3) its parent class name, (4) the name of the `View` class, (5) its relative location in the layout, (6) its resource ID, and (7) a hash of the text and content description.

To protect users' privacy and avoid capturing sensitive data, we hash the UI components text and content description. We also do not record any text entered by users.

7.3.1.2 Crowdsourcing App Usage Collection

Any Android device running our modified Android framework would be capable of collecting interactive user traces. But to do this at a large scale, we crowdsourced the task, deploying our recording framework on the PhoneLab [125] testbed as part of an IRB approved study. 99 users using Nexus 6 smartphones running Android 6.0.1 (CyanogenMod 13.0) participated in the experiment. We recorded how they interacted with different apps for 10 days between Nov. 1st and Nov. 10th, 2016. Importantly, most users of the testbed use their testbed device as their primary smartphone, which enables us to identify which apps are most popular and which usages are most common.

7.3.1.3 Identifying Common App Usages

From our PhoneLab experiment, we collected input event data from 99 users interacting with 531 apps. For this analysis, we chose the top 100 apps in terms of the total number of

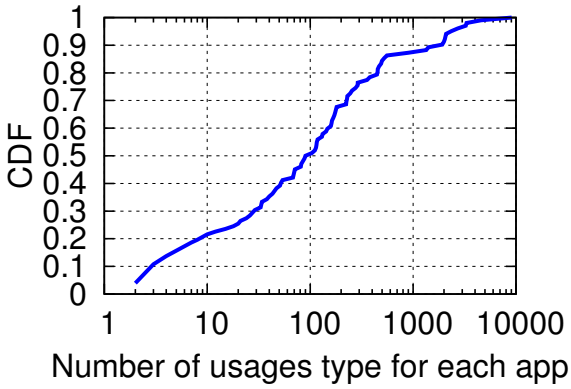


Figure 7.2: Number of distinct usages for top 100 apps.

interactions in our dataset. As illustrated in Figure 7.2, the number of distinct usages varies significantly across different apps. The app with largest number of usages is Facebook with 9082 different UI fingerprints that are touched by users 300K times in total.

Almost all the apps with a relatively large number of interactions but small number of usages are games and web-based apps. For the gaming apps, to display animation content and to be consistent across various app activities, all content is displayed in a custom-built `View`. For instance, for Candy Crush Saga, we only captured two usages in which a single custom View (`GameView`) is either scrolled or clicked by users. For web-based apps, all the content is displayed in a `WebView`. Since native Android widgets are not used by these two app types, we could not distinguish between different inputs, which limits our ability to replay users' interactions and create models based on usage.

Examining the frequency of the top interactions for each app allows us to identify common usages. Figure 7.3 shows the distribution of the frequency of interactions for each usage for six popular apps. As shown, 75% of total interactions for each app correspond to only 10 usages. This shows that we can capture how the app is being used and identify common user interactions by only considering a small number of usages. By crowdsourcing our usage data collection, we were indeed able to identify which apps are commonly used and which app usages are most common.

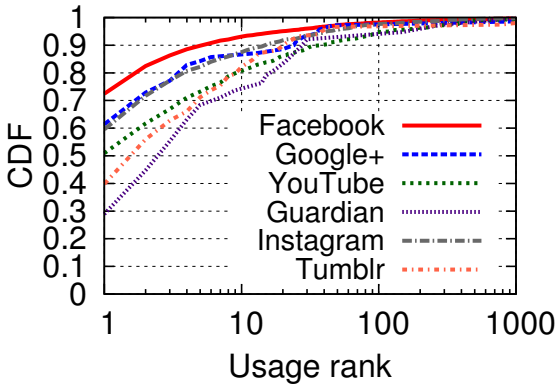


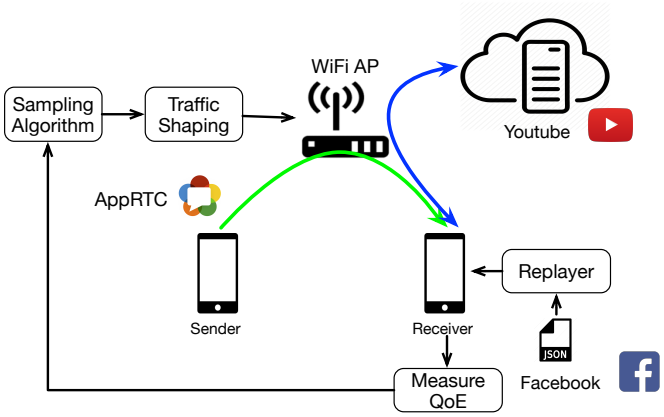
Figure 7.3: Distribution of the frequency of interactions for each usage for six popular apps.

7.3.2 Replaying App Usage Traces

Before we can measure QoE for the usages of interest and build a QoS-to-QoE model, we need a way to replay the app usage traces that were previously collected. To accomplish this, we built a new tool called *UsageReplayer*, which takes each individual trace as input, and replays the actions step by step to reach the particular usage of interest. The input to *UsageReplayer* is a `json` file, which contains the steps to reach the usage that have been extracted from each user input trace. Figure 7.4(b) shows an example `json` file to replay "scrolling on the search results of a keyword" in the YouTube app.

The `json` file consists of an array of actions. For each action, the fingerprint of its corresponding UI component is specified by the list of attributes that uniquely identify it. *UsageReplayer* measures the QoE metric for each action of interest if the value of `measure` attribute is `true`. *UsageReplayer* is an app-independent replay tool that is implemented using UIAutomator, which is an Android testing support library. This makes our approach suitable for black box testing—we do not need access to app source code.

We have published the source for *UsageReplayer* in Github [10]. Our sources include common usages with the 55 top apps collected from 99 users. This tool can be used to generate realistic smartphone application traffic by replaying and emulating how different users interact with different apps.



(a) Traffic shaping at WiFi AP applied to different types of apps including video conferencing, VOD, and interactive apps.

```
{"app": "YouTube",
"package": "com.google.android.youtube",
"name": "scroll_search_results",
"actions": [
  {"action": "click", "findby": {"id":
     "com.google.android.youtube:id/menu_search",
     "desc": "Search"}},
  {"action": "search", "findby": {"id":
     "com.google.android.youtube:id/search_edit_text"},
     "text": "election"},
  {"action": "scroll", "measure":true, "findby": {
     "id": "com.google.android.youtube:id/results"}}
 ]
}
```

(b) An example `json` file to replay "scrolling on the search results of a keyword" in the YouTube app

Figure 7.4: Our experimental setup for generating QoS-to-QoE mappings

7.3.3 QoE Measurement

Our goal is to build the QoS-to-QoE model for each usage by replaying it in a testbed where we can control the network conditions experienced by the app. But first, we must be able to measure QoE for each usage, which we do using the three techniques described below.

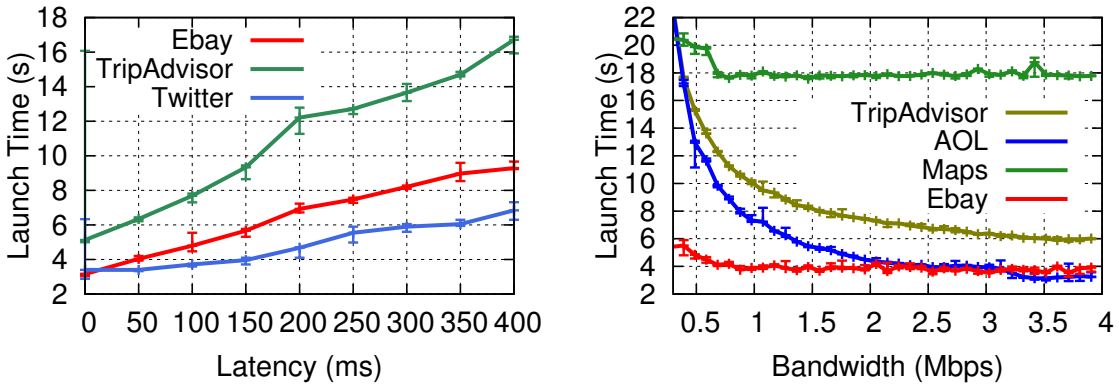
App-independent Android instrumentation. To capture how quickly apps respond to

each user input, we instrument the `onDraw` method in the `View` class. This method is invoked whenever any of the `View`'s child classes (*e.g.,* `TextView`, `ImageView`, and other UI components used by arbitrary apps) update their content. By passively monitoring the `onDraw` for a given user input, we can infer how long it takes for the app to respond and update the screen. To measure app responsiveness, we measure the time from user input to the last screen update via the `onDraw` event (we exclude UI update from Ads-related Views). This is analogous to the onload event for page load time in web browsing.

App responsiveness is a critical QoE metric [23, 153]. To the best of our knowledge, this is the first system that measures responsiveness for a large number of apps by passively monitoring changes to the screen. Differing from AppInsight [152], we instrument the Android framework as opposed to app binary. Therefore, our system does not need access to app binary and we do not need to instrument each individual app.

App source code instrumentation. In cases where the app is open source—such as AppRTC and ExoPlayer—we can instrument the source code directly. This allows us to record app statistics, debugging information, and QoE measures.

App-specific Android instrumentation. Different apps use different UI components and may have their own QoE metrics. To capture app-specific QoE metrics for these apps, we need to further instrument the Android framework in a way required by the particular app. For instance, YouTube uses Android's *throbber* widget when a video stalls. So to capture video stall events, we instrument the `ProgressBar` class. Another example is Skype. It exposes certain debugging statistics for developers using a `TextView`. Statistics shown include instantaneous QoS and QoE metrics. QoE Doctor [68] periodically records the UI tree to capture changes in app-specific UI components. In contrast, we directly instrument the corresponding classes of these components in the framework layer to monitor such changes. Our method is able to avoid the UI tree processing overhead and more accurately measure QoE metrics.



(a) Launch time monotonically increases with adding more latency.

(b) Launch time is not affected by increasing bandwidth when bandwidth is higher than a threshold.

Figure 7.5: Mapping (a) latency and (b) bandwidth to launch time.

7.3.4 QoS-to-QoE Mapping

To construct the mapping from individual QoS metrics to the corresponding QoE value, we vary one QoS metric at a time, keeping the other metrics fixed.

As depicted in Figure 7.4a, we emulate different network settings for latency, bandwidth, and loss rate through traffic shaping using `tc` at the WiFi access point. To emulate variable latency, we add extra delay to downlink packets using `netem`. To emulate bursty packet loss, we also use `netem`, which allows us to specify the percentage of packets to be randomly dropped, and how much dropping a packet should depend on its previous packet [29]. We run every app so that it can communicate with its own app server. Where necessary (*e.g.,* for collaborative apps such as those that offer video conferencing), we run multiple clients to mimic the operation of the app. We then measure its corresponding QoE value at the client using the techniques described above. For each network setting, we wait until QoE stabilizes, as there might be a delay during which the app tries to adapt to new network conditions.

We now describe how we generate this mapping for the usages collected from our user traces. We exclude YouTube, Hangouts, and Skype and generate their models separately, as their QoE metric is different from the rest of the usages.

7.3.4.1 Interactive apps

To create the model for the most common usages of each app, we picked the top three usages to replay for each app, which covers 74% of total user interactions. We excluded gaming apps and interactions for which we could not identify a UI component during replay—usually because a fingerprint failed to uniquely identify a UI component. We created replayable `json` files for 56 apps and 186 usages, which includes 83 scrolls, 47 clicks, and 56 launches.

For these 56 apps, we generate the model for three usage types: scroll, click, and cold start launch. For these usage types, app responsiveness—the time it takes for the app to update the screen—is the key QoE metric [23, 153]. To capture responsiveness, we use `onDraw` events as previously explained. We generate the model by replaying these usages under various QoS values. To construct the mapping, we vary one of the QoS metrics (*i.e.,* bandwidth, loss, and latency) at a time, while keeping the other metrics fixed.

To determine whether a usage is directly affected by latency, we compute Spearman's rank correlation coefficient between latency and app response delay. As shown in Figure 7.5a, we increase the downlink latency at 50ms granularity and measure its corresponding QoE value. Here, a high correlation coefficient indicates the monotonicity of the increase in app response delay when increasing end-to-end network delay. We find that for 49% of clicks and 85% of launches, the correlation coefficient between app response delay and latency is higher than 0.9. This indicates that most launches are latency sensitive. However, only 32% of the scrolling usages are latency sensitive. We find that for most of the scrolling usages with a low correlation coefficient, the app does not download the data while scrolling. Instead, it fetches the content when initially loading the `Activity`. However, latency-sensitive scrolls do lazy-loading and fetch content as the user scrolls.

To see how app response delay is affected by bandwidth, we increase the bandwidth, starting from 300Kbps, and measure its corresponding app response delay value. As shown in Figure 7.5b, we observe that QoE is not affected by increasing the bandwidth when band-

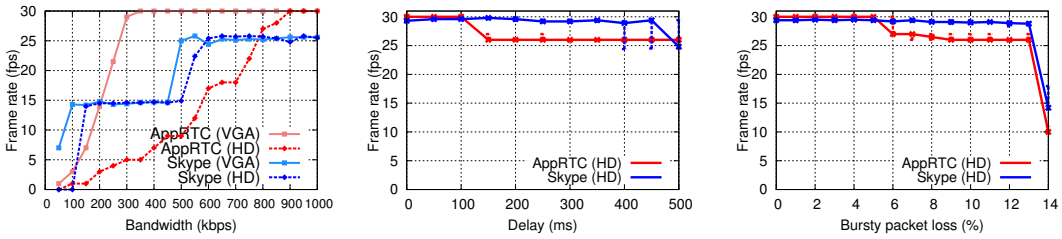
width is higher than a specific value. We find that these values—which specify the bandwidth requirement of each usage—are in fact small (median bandwidth is around 2.2Mbps for launch and 1.5Mbps for click). This is attributed to the fact that most of mobile apps' uplink and downlink transfers are small (less than 100KB [96]), and under high bandwidth conditions, latency plays a more important role in determining short-lived flows' performance than bandwidth does.

As we have shown, the QoS requirement of each usage can be derived from its QoS-to-QoE model. In §7.5.1, we will show how leveraging this information can help ISPs improve QoE of latency sensitive usages.

7.3.4.2 Video streaming and video conferencing apps

Video conferencing and on-demand video streaming are two popular types of apps with different QoE metrics and requirements. For video conferencing, frame rate and video/audio delay are the key QoE metrics and have direct relationships with user satisfaction [185, 189]. For video streaming, video bit-rate and rebuffering frequency are the key QoE metrics [103, 114, 57]. For video conferencing, we generate models for two popular apps: AppRTC [49] and Skype. AppRTC is a video conferencing app developed by Google. It uses Chrome's native WebRTC implementation and shares the same WebRTC code base as Google Hangouts. For on-demand video streaming, we use the ExoPlayer library [12]. It provides a pre-built video player for Android using DASH and is currently used by YouTube and Google Play Movies [13]. Using ExoPlayer, we can play three state-of-art HTTP-based adaptive streaming schemes: YouTube DASH, Apple HLS, and Microsoft Smooth Streaming.

To minimize disruption, both apps adapt their QoE to variations in QoS. AppRTC and Skype adapt their encoded and decoded frame rate to respond to available bandwidth. As shown in Figure 7.6, various QoS metrics affect QoE differently. Since both Skype and AppRTC use Forward Error Correction (FEC) techniques [185] they can tolerate some amount



(a) Mapping bandwidth to frame rate for VGA and HD video quality.

(b) Mapping delay (added by AP) to frame rate for VGA and HD video quality.

(c) Mapping packet loss to frame rate for HD video quality.

Figure 7.6: Mapping various QoS metrics to frame rate (QoE) for AppRTC and Skype

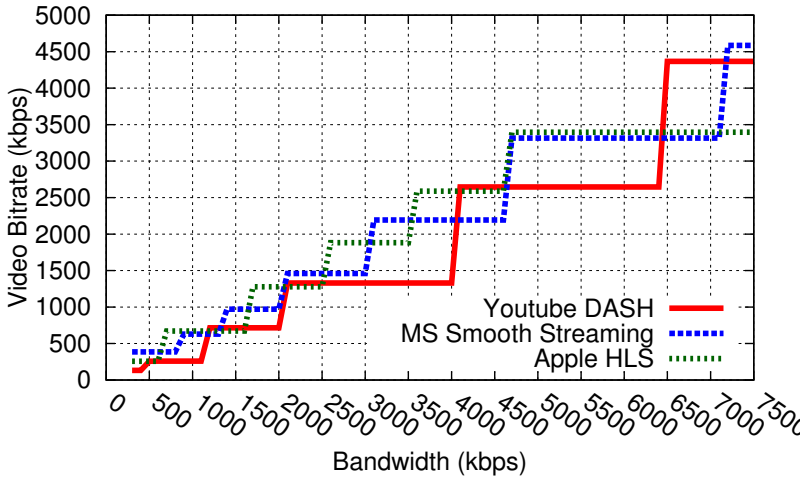


Figure 7.7: Mapping bandwidth (QoS metric) to video bitrate (QoE metric) for Youtube, Microsoft Smooth Streaming, and Apple HLS

of packet loss—up to 5%. However, both are highly sensitive to bandwidth variations.

For all three metrics, we observe that changes in QoS do not necessarily lead to changes in QoE. For example, video frame rate changes only at certain transitions in QoS. We made the same observation for all the video steaming schemes shown in Figure 7.7. Here the relationship between QoS and the QoE metric (video bitrate) is even more discrete [1]. This is particularly important to consider when performing traffic management, since the impact of changing bandwidth on app QoE is important for network operators. In §7.5.2 we show how ISPs can tune QoS to control QoE by using the QoS-to-QoE mapping for the

[1]These mappings are consistent across different videos, as video streaming services usually transcode uploaded videos into a specific set of bitrates.

Algorithm 2: Adaptive sampling of QoS metric space

```
1  Procedure SAMPLE ($n$-dim space $R$)          /* $n$ QoS metrics with arbitrary range $r$ */
2      NewSubSpaces ← {...}
3      for each r_i do
4          if r_i ≤ Thresh_i then
5              R_i1, R_i2 ← divide r_i by 2
6              NewSubSpaces.append(R_i1, R_i2)
7          end
8      end
9      if len(NewSubSpaces) = 0 then
10         return
11     else
12         for each R_i in NewSubSpaces do
13             BadQoESamples ← 0
14             for each Edge e_j do                /* Each Space R has 2^n edges */
15                 if QoE (e_j) =Bad then
16                     BadQoESamples ← BadQoESamples +1
17                 end
18             end
19             if 0 < BadQoESamples/2^n < 1 then
20                 SAMPLE (R_i)
21             end
22         end
23     end
```

corresponding app.

7.3.5 Adaptive Sampling of the QoS Metric Space

While we have shown how to map from individual QoS metrics to the corresponding QoE value, constructing a precise model

$$QoE = f(bw, delay, loss\_rate)$$

requires emulating all combinations of QoS values. However, as QoS metrics are continuous variables, experimenting with all possible combinations is impractical. We propose a sampling technique to find important combinations of QoS values. We argue that we can map QoE values to a limited set of QoE classes. In fact, customer satisfaction models typically follow a threshold-based approach to distinguish between various levels of customer satisfaction and dissatisfaction. For example, if frame rate is above a threshold, users may

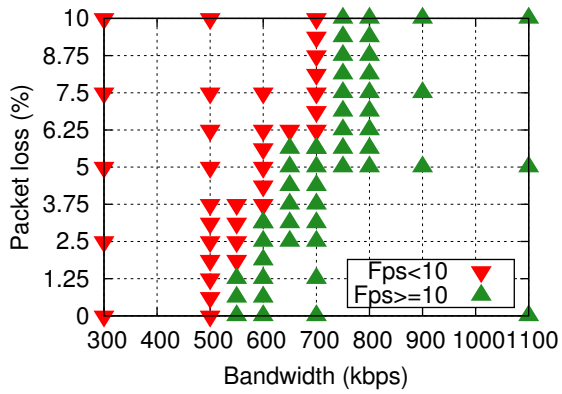


Figure 7.8: Sampled QoS values based on Algorithm 2

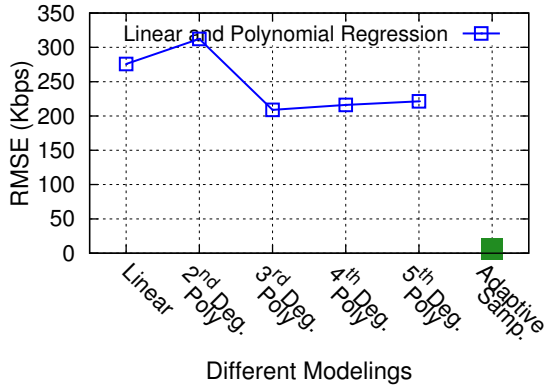


Figure 7.9: Compare accuracy of adaptive sampling with regression-based modelings.

not notice any further improvement. Then, we can selectively increase our sampling of the QoS metric space close to the borders of different QoE classes.

We describe our algorithm in Algorithm 2. In this algorithm, to identify the boundary between different classes of QoE, the algorithm reduces the search space by half and if it observes samples from different QoE classes in each sub-space, it recursively calls the algorithm on that sub-space. For simplicity, we present the version of our algorithm for the case where we have a *thresholded* model for QoE [67, 51] with two classes —*Good* or acceptable and *Bad* or unacceptable—and n QoS metrics. But the algorithm is easily extensible to more than two QoE classes. We demonstrate the result of sampling for AppRTC in Figure 7.8. For simplicity, two QoS metrics are sampled: bandwidth and packet loss. As shown, our sampling algorithm is able to clearly identify the boundary between the two classes of QoE.

7.3.6 Evaluation

We evaluate the accuracy of the models generated by the adaptive sampling technique by comparing it with mappings that use linaer and polynomial regression. Specifically, we compare our technique with Prometheus [51], which uses linear regression (*i.e.,* LASSO regression) to map network traffic features to the binary classification of QoE. We consider the model we generated for a Microsoft Smooth Streaming video (Figure 7.7) to compare the accuracy of our adaptive sampling technique with linear and 2nd to 5th degree polynomial regression. To train the regression-based models, we generate the same number of randomly selected bandwidth and video bitrate samples. To measure the accuracy in terms of root mean squared error (RMSE), we used another set of random samples (20% of the size of the samples we used for training). Figure 7.9, shows that our model provides 98% and 97% higher accuracy for predicting the video bitrate than linear (*e.g.,* Prometheus) and 2nd degree polynomial regression, respectively. This is attributed to the fact that due to the complex interaction between app protocol and network conditions, we may not be able to generate the QoS-to-QoE mapping using linear or polynomial models.

7.3.7 Impact of System-level QoS Metrics

While we study and model the impact of network-level QoS metrics on app QoE, system-level QoS metrics (*e.g.,* CPU load, GPU, RAM) may also affect QoE. For instance, poor GPS signal can increase launch time of an activity tracker app. To understand how robust the models are to common system-level QoS variations and device hardware differences, we generated the models in two scenarios: (1) device hardware differences, for which we experimented on Galaxy S3 (2012) and Galaxy S7 (2016), and (2) resource contention from background apps, for which we used a synthetic background program to increased CPU load by 20% (the maximum increase from background apps [69]). We performed experiments for both video streaming and video conferencing, but did not find any change in the generated model. This suggests that the models generated for the apps con-

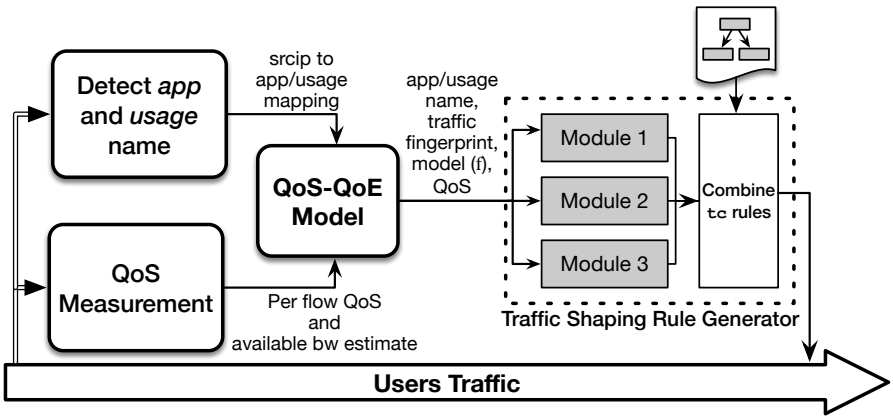


Figure 7.10: QOEBOX architecture.

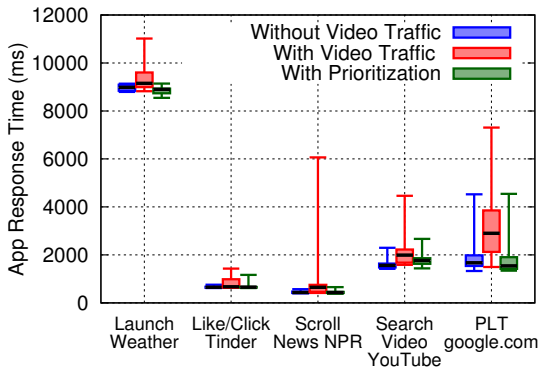


Figure 7.11: App response delay of five usage types under three scenarios: without concurrent video traffic, with video traffic, and with prioritization.

sidered in this chapter are relatively robust to common system-level QoS metric variations.

We do admit that for certain apps, e.g., some gaming apps, the QoE can be affected by system-level QoS metrics such as GPU and CPU. Such models that consider both system-level and network-level QoS metrics can be utilized in OSes (*e.g.,* OS-level scheduler) and are a part of our future work.

7.4 QOEBOX: QoE-Aware Traffic Management

To optimize the use of their network, network operators may utilize various traffic shaping techniques. For example, an ISP may throttle flows traversing a congested link. Or, it may attempt to apply limits to certain traffic categories, such as streaming video [174].

To respond to different app demands, we argue that network operators cannot treat all traffic equally. For example, in the face of congestion they should attempt to allocate network resources to minimize the degradation in QoE experienced by users. As shown by previous studies [113, 124], when QoE drops below a certain threshold, users become frustrated. They may respond by quitting the app or abandoning the service.

In this section, we demonstrate the practical utility of per-app QoS-to-QoE models for addressing these problems. We design, implement, and evaluate QoEBox, a QoE-centric traffic management framework. QoEBox is a proxy solution that can be installed on any Linux-based middlebox located on a network path carrying all traffic for one or more users. QoEBox is transparent to both user-facing apps and backend servers. It does not need to communicate with end-hosts to get instantaneous user-perceived QoE. Instead, it relies on per-app QoS-to-QoE models, which are generated offline, to applies various traffic shaping techniques to optimize the use of network and improve end-user QoE.

As illustrated in Figure 7.10, QoEBox consists of three main components: app name and usage detection, QoS measurement, and traffic shaping rule generator.

7.4.1 App Name and Usage Detection

As traffic from one or more devices is routed through QoEBox, the first step is to detect app name and usage for each flow. This allows the system to find its corresponding model and apply the appropriate QoE-aware traffic shaping policies.

To map flows to apps, network operators can use existing deep packet inspection tools such as nDPI [28] or other app traffic classification techniques [186, 182]. However, we face two unique challenges not addressed by existing techniques. First, we need to detect usage as well as app name, as different usages within the same app may have a different QoS-to-QoE models. For instance, searching for videos in YouTube is latency sensitive, while watching videos is bandwidth sensitive. Second, we need to detect the app name and usage as early as possible. This is particularly important for latency-sensitive traffic, which

tends to be short-lived. This gives QoEBox only a short amount of time to identify the usage and apply the proper traffic shaping policy.

Our implementation of QoEBox uses a DNS-based technique to infer the app name and usage. Our technique is based on the observation that for each usage, the set of domain names used by uplink traffic is unique to that usage. For example, when an app is launched, it may send data to various tracking and advertisement services, whereas for scrolling, it may only fetch the content that is going to be shown to user. To create a fingerprint for each usage, we replay usages on different devices at different times of day.The fingerprint for each usage will be the intersection of the sets of domains observed across different runs.

To evaluate the accuracy of app name detection, we applied the technique on a dataset of 6 smartphone users' network traffic. These traces include packet payloads and packet-app mappings of 59 apps (31K TCP conn.) and were collected over a period of 66 days (May to July 2016) as a part of an IRB approved user study. In this dataset, among all the connections, only 4.3% do not use domains.

We split the data into a training and a testing data set, where the training set covers the first 36 days of the data collection period and corresponds to 82% of connections, and the testing set contains the rest of the data. Then, the app name of each individual session is predicted based on its fingerprint generated in the training phase. Out of 5.6K connections in the testing data, it correctly identifed the app name of 70% of connections, and for 63% of the apps, the app name of all their corresponding connections are correctly predicted, which shows that they use unique domains for fetching their content. We find that almost all of the apps with low prediction accuracy are Google apps (*e.g.,* Gmail, Calender, Maps, etc.) that use the same set of domain names.

Since the collected network traces are only labeled with app name, to evaluate the accuracy of usage detection, we chose the top 20 apps in our app usage dataset and for each app, we create fingerprints for the top 5 usages. For 27 usages, the apps do not establish any connection, which means that the content is downloaded before the user input. We exclude

these 27 usages from our study. We find that for the launch usage type, all the fingerprints are unique. Overall, 64 unique fingerprints are created for 73 usages. Out of 73 usages, 56 usages (76%) are mapped to a unique fingerprint. Other usages' fingerprints are not unique within their app. Here, we present a simple technique with reasonable accuracy, as being able to detect app names and usages is one of the requirements of this system. We plan to incorporate other features to improve the accuracy in our future work. We discuss the implications of not being able to map users' traffic to their corresponding model in §7.5.

Existing app identification techniques fail to provide fast detection, as they use payload [28, 186, 182], or packet size [52] to extract app fingerprints. The main benefit of our technique is the fact that using domains as the feature will allow us to identify usages only from SYN packets. Using these fingerprints that are created offline, QoEBox can map a set of IPs to their corresponding app/usage name.

7.4.2 QoS Measurement

To infer end-user QoE for a given app, we must be able to measure the QoS of its flows in order to apply its QoS-to-QoE model. This includes an estimate of bandwidth consumed by app, packet loss, latency, and an estimate of available link bandwidth.

Since not all network operators have visibility at the network edge, the ability to infer QoS depends on the metric and protocol specification. For example, some QoS metrics such as UDP packet loss can be measured in the core network only if the protocol exposes some information such as sequence number.

To overcome these challenges, we leverage existing tools and techniques to measure bandwidth, packet loss, and delay from passive analysis of user traffic [96]. For instance, to infer packet loss and delay from TCP traffic, we keep track of sequence/ack numbers and TCP handshake RTT. For UDP traffic, if the protocol includes timestamp and sequence number, it is possible to estimate delay and measure packet loss. For instance, the Real-time Transport Protocol (RTP)—a popular protocol for real-time applications and used by

WebRTC [49]—includes both timestamp and sequence number in the header. Even if the payload is encrypted we can still measure TCP and UDP throughput.

While measuring bandwidth, loss, and latency for TCP connections and UDP streams, QoEBox also measures available link bandwidth and reports it to the traffic shaping rule generator module. Available bandwidth is needed when multiple bandwidth-intensive usages are detected and competing for network resources. In that case, QoEBox can leverage the model to optimally allocate the available bandwidth to competing flows. In §7.5.2, we describe the design, implementation, and evaluation of a QoE-aware bandwidth allocation scheme that leverages this information. To estimate available bandwidth, we adopt the technique described in [87]. By passively monitoring the apps' traffic and measuring aggregated throughput of *all* TCP flows that have transferred enough bytes to exit slow-start, QoEBox can estimate the achievable bandwidth when the link is saturated.

7.4.3 Traffic Shaping Rule Generation

QoEBox allows network operators to implement various QoE-aware traffic management schemes and include their implementation as a *module*. As input, each module takes the model, as a function, per-flow QoS information, app and usage names, and their corresponding traffic fingerprint. As output, the module generates a set of `tc` rules for a *class* of traffic. In the generated `tc` rules, `filters` specify the traffic that should be processed by each class. Here, the traffic fingerprint of each usage can be included in the filters.

Modules can classify traffic based on various attributes, including app name and app type (*e.g.,* video streaming), and one module can apply its shaping on a class of traffic that is classified by another module. Thus, we have a notion of ordering between classes. We will showcase an example of this ordering in §7.5, in which a module first classifies the traffic as latency sensitive or bandwidth intensive, then another module applies QoE-aware bandwidth allocation only on traffic belonging to the bandwidth intensive class. To enforce ordering among different modules, QoEBox takes the relation between the modules as

input and uses Hierarchical Token Bucket (HTB) qdisc to combine the generated rules from different modules.

QoEBox is designed and implemented as a middlebox that can be installed on any Linux-based machine. Depending on where QoEBox is installed (*e.g.,* cellular base-station or home WiFi access point), it can measure and control different QoS metrics. Thus, as a result of the modular design, network operators can implement and include different modules to control QoS and enforce various QoE-aware traffic management schemes on different platforms and networks.

7.5 Case Studies

We design, implement, and evaluate two modules for WiFi access points, as case study: a traffic classification and prioritization module, and QoE-aware bandwidth allocation module. Both make a direct use of the models generated for various types of apps. For evaluation, we cross-compile QoEBox and these modules as a package with OpenWrt SDK for Chaos Calmer (15.05.1) release. We evaluate QoEBox on a TP-Link Archer C7 WiFi router with OpenWrt 15.05.1 and we consider home network scenarios where the number of users typically ranges from 1 to 10.

7.5.1 Classifying and Prioritizing Different Traffic Types

Our goal is to properly allocate resources and satisfy usages' various QoS requirements by applying traffic shaping policies while managing competition between traffic from different usages. To do this, this module first classifies the traffic based on the app and usage's model. Traffic from different user interactions may behave differently under different network conditions. For instance, normally video streaming is delay tolerant. But at low bandwidth, as the video chunk sizes become smaller, it may also become sensitive to increases in latency. Thus, this module takes the current QoS values as an input to classify the flows.

As a result of classification, apps and usages with the same requirements will be assigned to the same class. This separation and control is necessary, as traffic from different classes can interact poorly and adversely affect each other's QoE. The poor interaction between different classes of traffic can be caused by the fact that each usage has its own QoS requirements. As a result, they tend to leverage different protocols that satisfy their needs. For instance, video streaming apps require high bandwidth to provide high quality video to users. To achieve this goal, video streaming apps use TCP, which tries to aggressively consume all the available bandwidth. However, in the presence of other classes of traffic, this strategy can adversely affect the QoE of other apps.

QoEBox classifies flows into three classes:

1. **Latency sensitive traffic** that includes various usage types, such as click, launch, and scroll in interactive apps;

2. **Bandwidth and latency sensitive traffic** that represents video conferencing apps; and

3. **Bandwidth intensive traffic** such as video streaming, foreground app installation, and file downloads.

and uses HTB qdisc in `tc` to generate the classes.

We first focus on characterizing the interaction between these three classes of users' traffic. To characterize how bandwidth intensive traffic affects the other two traffic classes, we conduct the following experiment. First we measure the QoE of 5 devices replaying different types of latency sensitive usages. We repeat this 20 times for each usage. Then we add three more devices that play a YouTube video, representing bandwidth intensive usage. We compare the app response delay for latency sensitive usages with and without the presence of video streaming traffic. We repeat this experiment for video conferencing with AppRTC which represents bandwidth and latency sensitive usage. We then compare video quality of video conferencing with and without the presence of video streaming traffic.

Since YouTube uses TCP, it can consume all available bandwidth. We observe that this behavior affects video conferencing and interactive latency sensitive apps in two different ways. Figure 7.11 shows that when video streaming traffic co-exists with different latency-sensitive usages, the median, 75th, and 95th percentile of the response time for all 5 usage types increases. For example, median PLT of `google.com` rises by 73%, while 95th percentile of app response time for scrolling in NPR increases by 964%. This translates into between 1 and 6 s of extra delay. As explored in previous work [128], this increase in app delay is caused by bursty video traffic that increases queuing delay at the router buffer and corresponding end-to-end latency.

For video conferencing apps, Figure 7.12 shows that when other users are watching YouTube, median frame rate drops from 24 fps and 27 fps to 2 fps and 7 fps for AppRTC and Skype, respectively. For AppRTC, although the SCTP protocol aggressively tries to obtain more bandwidth, it loses the competition with TCP and cannot deliver all frames successfully. In addition, additional delay caused by queuing in the router leads to high frame drops at the receiver. Because video conferencing apps cannot tolerate delay, late packets are dropped.

Previous studies have addressed poor interaction between different traffic types by prioritizing traffic from latency-sensitive apps. QoEBox prioritizes traffic classes as follows: (1) Latency-sensitive traffic, (2) Latency and bandwidth sensitive traffic, (3) Bandwidth-intensive traffic, and (4) Background traffic, which is both latency and bandwidth tolerant. We used `prio` qdisc in `tc` to enforce these priorities.

The effects of traffic prioritization are shown in Figures 7.11 and 7.12. Figure 7.11 shows that latency-sensitive usages recover the performance they achieve without concurrent video traffic. Figure 7.12 shows that video frame rate for the video conferencing app also improve to their values without concurrent video traffic. By prioritizing the traffic of latency sensitive usages, these usages can automatically receive the amount of bandwidth they need, thus satisfying their bandwidth requirement, as explained in §7.3.4.

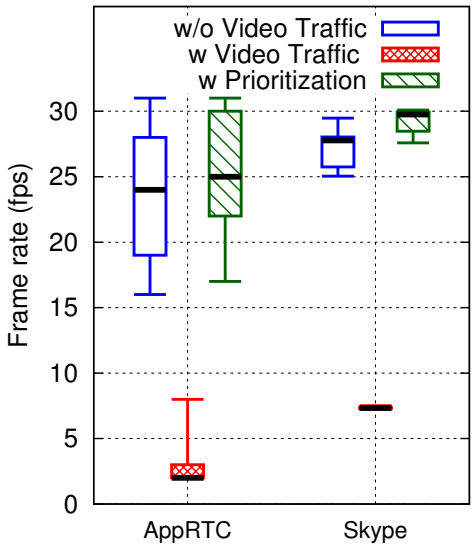


Figure 7.12: Frame rate of AppRTC and Skype w/o and with video traffic, and with prioritization.

For flows with unknown models—which is caused by inaccurate app name and usage detection—we ensure that their performance will not be affected by QoEBox. To achieve this goal, QoEBox first tries to detect whether a flow with an unknown model is short-lived or long lived. If it is short-lived, it will be assigned the same priority as latency and bandwidth sensitive traffic, otherwise, it will be assigned the lowest priority. In terms of bandwidth, it will be allocated an equal share of the available link's bandwidth.

Although prioritizing the traffic has been extensively explored in previous studies, we are the first to demonstrate its effectiveness on improving QoE of various types of apps and usages.

7.5.2 QoE-Aware Bandwidth Allocation

One observation arising from the model we built for video streaming and conferencing apps is that a change to QoS does not necessarily lead to a change in QoE. For all the three adaptive video streaming schemes in Figure 7.7, the video bitrate is discrete. One immediate implication of the model is that allocating more bandwidth to video streaming apps does not necessarily improve the video bitrate. Moreover, applications do not seem to

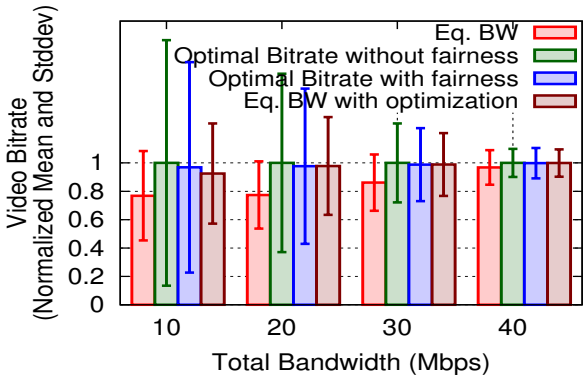


Figure 7.13: Video bitrate of 10 users streaming video at random times and duration.

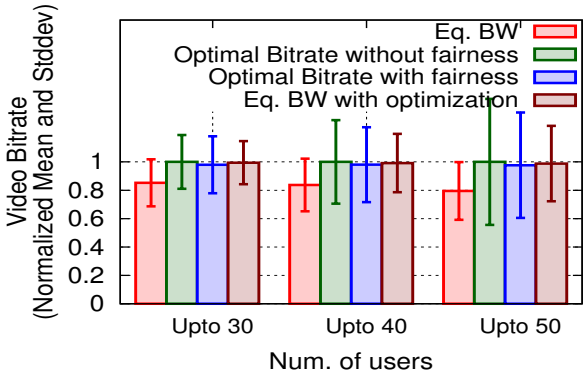


Figure 7.14: Simulation results of different number of users streaming video under 100Mbps bandwidth.

control the amount of bandwidth they consume. This is due to use of TCP at the transport layer, which may try to download chunks as fast as possible and consume a large amount of bandwidth, and that amount of bandwidth might be more than enough for the selected bitrate.

To understand this effect better, we did an experiment with YouTube. Based on the YouTube model in Figure 7.7, 2.5Mbps and 3.5Mbps will lead to the same bitrate. But the app does in fact consume its total allocated bandwidth. YouTube consumes the allocated bandwidth to download video chunks faster, but the extra bandwidth does not result in a higher bitrate. Thus, allocating the optimal amount of bandwidth to each user is necessary.

To motivate the need for limiting the bandwidth of each user, we run an experiment with 10 devices simultaneously streaming a 10 min HLS video. We first limit the total

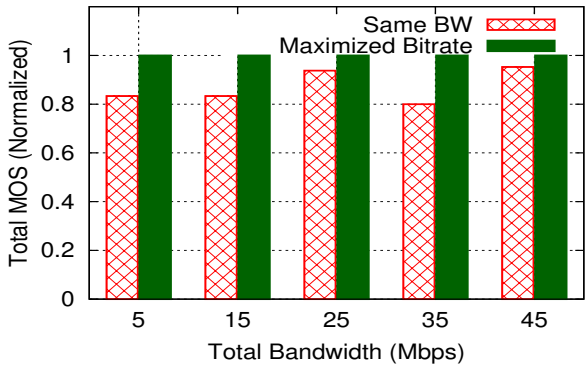


Figure 7.15: Total MOS of 10 users streaming video and downloading a file at the same time.

bandwidth to 20Mbps and compare the mean video bitrate and rebuffering duration with the case where each device is individually limited to 2Mbps. We find that when limiting the bandwidth of each user separately, the mean video bitrate drops by 10%. However, the mean rebuffering time improves from 18 s to no rebuffering. We attribute this to the fact that due to the buffering, downloading consecutive video chunks exhibits an ON-OFF pattern. When multiple capacity-based ABR clients are streaming videos simultaneously, the overlap between these ON-OFF patterns may cause devices to overestimate the available bandwidth. This causes them to switch to a too-high bitrate, which eventually leads to rebuffering events.

These observations lead us to derive a new bandwidth allocation scheme to improve overall QoE of video streaming apps. To allocate the bandwidth efficiently across different apps with different models, we formulate the problem as an optimization problem. We first use the model to find the *minimum bandwidth required* for each QoE class (for video streaming, QoE classes correspond to discrete bitrate values). To account for fluctuations in network bandwidth or any inaccuracy in the model, we add extra bandwidth ϵ[2] to the minimum value. We then form the function f to map discrete bandwidth values b to their corresponding bitrate value v. Then, for each user, we need to choose from available discrete bandwidth values to maximize the overall bitrate with the constraint that the total

[2]for our experiments, we choose 100Kbps as ϵ

allocated bandwidth must be less than available bandwidth.

This problem is equivalent to the *multiple-choice knapsack problem (MCKP)* [107]. The bitrate v_{ij} corresponds to the value of bandwidth j for app i and we need to choose exactly one bandwidth value for each device. We can also incorporate additional constraints reflecting common ISP policies—limiting the total bandwidth of a particular app, or capping users' bandwidth based on their service plan.

One issue with MCKP is that it does not consider fairness, so the users of the same app may be assigned different bandwidth values. To enforce fairness across the users of the same app, we modified the MCKP formulation as follow to make the users of the same app receive equal bandwidth:

$$\text{maximize} \sum_{i=0}^{n} \sum_{j \in S_i} V_{ij} x_{ij}$$

$$\text{subject to} \sum_{i=0}^{n} \sum_{j \in S_i} B_{ij} x_{ij} \leq \text{Total}BW,$$

$$\sum_{j \in S_i} x_{ij} = 1, i = 1, \ldots, n \quad x_{ij} \in \{0, 1\}, i = 1, \ldots, n, j \in S_i$$

Where:

- n is the number of apps—instead of users in MCKP;
- S_i is the set of bandwidth values available to choose for app i;
- $V_{ij} = k_i \times v_{ij}$ where k_i is the user count for app i;
- $B_{ij} = k_i \times b_{ij}$ is the total bandwidth allocated to the users of app i.

Various algorithms are available to solve the MCKP. We used a dynamic programming formulation explained in [107] to solve MCKP. Solving MCKP with dynamic programming requires finding optimal bandwidth allocations (*i.e.,* B_i for app i) for a *range* of input values, *i.e.,* available bandwidth values and apps. Using these precomputed solutions, we

can update the apps' bandwidth value in response to the changes in available bandwidth, without extra computational overhead. We use HTB qdisc to limit the bandwidth of each app to its corresponding precomputed bandwidth B_i.

To evaluate how much our optimization algorithm can improve overall video bitrate, we did an experiment with 10 devices. Each device plays a video from a random app at random times and for a random duration. We perform this experiment for 1 hour. Then we compare mean video bitrates of devices using three different bandwidth allocation approaches: (1) same bandwidth for each device, (2) optimal bandwidth without fairness, and (3) optimal bandwidth with fairness. We show the results under four different bandwidth values in Figure 7.13. Optimal bandwidth without fairness improves mean bitrate by up to 30%, and with fairness by up to 26%. However, as can be seen, there is a high standard deviation in both approaches, meaning that MCKP may allocate high bandwidth to some users, while other users may be allocated a very low bandwidth.

To reduce this disparity, we modify the optimization formulation in the following way. First, we equally divide and allocate the available bandwidth to all the users ($\bar{b}$). Then for each user, based on its model, we find $\bar{b}$'s corresponding bitrate value v_i and then the minimum bandwidth required for v_i (b_i). Here if $b_i < \bar{b}$, we can potentially use all the extra bandwidth of $B_{\text{extra}} = \sum_i \bar{b} - b_i$ to improve video quality. To do so, we formulate the problem as a MCKP with B_{extra} as the total available bandwidth and optimally allocate this extra bandwidth to users that can use it to switch to higher bitrates. As can be seen in Figure 7.13, compared with optimal bandwidth with and without fairness, this approach can achieve the lowest standard deviation in exchange for 1 to 7% lower bitrate.

We also performed a simulation of public WiFi networks, where the number of users can be higher (up to 50). As in our home WiFi network experiment, users can start watching videos at random times and for a random duration. As shown in Figure 7.14, with 100Mbps bandwidth, we can achieve up to 25%, 22%, and 24% higher average bitrate for the three proposed bandwidth allocation schemes.

Here we formulate the problem for video streaming apps and we maximize a single QoE metric (*i.e.,* video bitrate). In case there are multiple bandwidth intensive apps with *different* QoE metrics (*e.g.,* downloading a file in foreground and video streaming), first we need to map all various QoE metrics to a *single* metric. To do so, we can utilize existing models and subjective quality evaluation methods that provide the mapping between different app-specific objective QoE metrics to app-independent MOS (Mean Opinion Score). To evaluate our proposed bandwidth allocation scheme in such scenario, we did an experiment with 10 devices: 8 devices streaming YouTube videos and 2 devices are downloading a 10MB file[3]. We use the models that map video bitrate to MOS from [110] and download time to MOS from [81]. Figure 7.15 shows the results under five different bandwidth values. As can be seen, our proposed scheme improves the overall MOS by 25%.

7.6 Conclusion

In this chapter, we propose offline generation of per-app models mapping app-independent QoS metrics to app-specific QoE metrics. By building QoEBox, a QoE-based traffic management framework, we show how network operators can utilize these QoS-to-QoE models to optimally allocate the resources between various types of apps and improve end-user QoE. We design, implement, and evaluate two direct applications of the model, as a QoEBox modules for WiFi access points: a traffic classification and prioritization and an optimal fair QoE-aware bandwidth allocation scheme.

[3]Traffic of mobile video is four times the traffic of app download and file sharing [11]

CHAPTER VIII

Conclusion

Three main projects are discussed in this dissertation, aiming to answer key questions of (1) how to provide visibility into the end-user application performance, and (2) how this visibility can help in detecting, diagnosing, and addressing performance problems on mobile devices.

We answer the first question by designing a system, called *Mobilyzer*, that efficiently provides controllable and accurate network measurements in the mobile environment. We show that our system is efficient, easy to use and supports new measurement experiments in the mobile environment. We show that using *Mobilyzer*'s support for coordinated measurements among large numbers of vantage points; we can evaluate the performance of Internet-scale systems. This includes identifying and diagnosing cases of CDN inefficiency, characterizing and decomposing page load time delays for web browsing, and evaluating alternative video bitrate adaptation schemes in the mobile environment. Further, in the second project, we use this platform to study how MPTCP, a new transport layer extension, performs in practice. Combined with passive measurement techniques, we study MPTCP's overhead and its impact on end-user application performance. We introduce new system designs to address the limitations, which were revealed by our measurement study.

We believe this platform can be utilized by the research community to characterize the performance of newly proposed application or transport layer protocols (*e.g.,* QUIC) and

understand how they perform on real users' devices. Additionally, the server-side measurement scheduling component (*i.e.,* global manager) can be further improved to include an interface that supports a flexible set of queries. Using this interface, researchers, operators and providers can ask questions like *"What is the average round-trip latency of the users in a certain area to a specific server?"*. To answer these types of queries, the global manager can convert the query to a set of measurement tasks, predict resource availability, and optimize measurement deployment. In case there aren't sufficient resources to conduct the measurements, the global manager can rely on historical data, collected from previous measurements, to provide an approximate answer to the query.

Since it is impractical to use active probes to measure application performance at scale over all the devices and applications, in the third project we show how various entities, including wireless carriers and ISPs, can infer end-user application performance by passively monitoring users' traffic. Our proposed approach for inferring QoE corresponding to a traffic flow is to rely on models that can map the flow's QoS metrics to the corresponding app's QoE metrics. To demonstrate how network operators can utilize these models to optimally allocate the resources between various types of apps and improve end-user QoE, we design a QoE-centric framework and two traffic management schemes, as modules, for the framework: a traffic classification and prioritization module and an optimal fair QoE-aware bandwidth allocation module. We show that these traffic management schemes can successfully improve the QoE metrics that reflect user-perceived performance. Designing and developing additional traffic management schemes can be an interesting direction for future work. Additionally, conducting user studies to evaluate how much our proposed techniques can improve real users' QoE is left for future work.

In all these projects discussed in this dissertation, we devise novel active and passive measurement approaches to conduct measurements of end-user transport and application layer performance. Using these systems, we characterize the performance of Internet-scale systems and protocols and identify their limitations by conducting in-lab and crowd-

sourced measurement studies. Motivated by our findings, we propose new systems and approaches to address the identified shortcomings. We believe the methods and systems introduced and presented in this dissertation can be utilized by future research to better understand how various protocols and techniques perform in the highly dynamic mobile environment.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] The 16 chains with the best free wi-fi, ranked - cnet. https://www.cnet.com/pictures/chains-with-the-best-free-wifi-ranked/.

[2] Akamai Secure Token. http://help.ooyala.com/video-platform/concepts/akamai_token.html.

[3] Android platform versions. https://developer.android.com/about/dashboards/index.html.

[4] AT&T to Launch Mobile 5G in 2018. http://about.att.com/story/att_to_launch_mobile_5g_in_2018.html.

[5] Building zero protocol for fast, secure mobile connections. https://code.facebook.com/posts/608854979307125/building-zero-protocol-for-fast-secure-mobile-connections/.

[6] Chrome's Telemetry performance testing framework. http://www.chromium.org/developers/telemetry.

[7] Cisco visual networking index: Forecast and methodology, 20162021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html.

[8] Cisco visual networking index: Global mobile data traffic forecast update, 20162021 white paper. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[9] Connectivitymanager — android developers. https://developer.android.com/reference/android/net/ConnectivityManager.html.

[10] *UsageReplayer* github repository. https://github.com/AndroidUsageReplayer/AndroidUsageReplayer.

[11] Ericsson mobility report, june 2016. https://www.ericsson.com/assets/local/mobility-report/documents/2016/ericsson-mobility-report-june-2016.pdf.

[12] ExoPlayer. https://google.github.io/ExoPlayer.

[13] ExoPlayer: Adaptive video streaming on Android - YouTube. https://www.youtube.com/watch?v=6VjF638VObA.

[14] Google Developers, Critical Rendering Path. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/.

[15] Google Web Page Replay Tool. https://github.com/chromium/web-page-replay.

[16] Half of YouTube's traffic is now from mobile - CNBC.com. http://www.cnbc.com/id/102128640.

[17] High performance browser networking, o'reilly. https://developer.android.com/about/dashboards/index.html.

[18] Hulu: Internet speed requirements for streaming hd and 4k ultra hd. https://help.hulu.com/en-us/requirements-for-hd.

[19] IEEE SA - 802.11e-2005 – part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications - amendment 8: Medium access control (mac) quality of service enhancements. https://standards.ieee.org/findstds/standard/802.11e-2005.html.

[20] Internet Speed - Android Apps on Google Play. https://play.google.com/store/search?q=network\%20tools&c=apps&hl=en.

[21] iOS: Multipath TCP Support in iOS 7. https://support.apple.com/en-us/HT201373.

[22] KT's GiGA LTE. https://www.ietf.org/proceedings/93/slides/slides-93-mptcp-3.pdf.

[23] Measure performance with the rail model. https://developers.google.com/web/fundamentals/performance/rail.

[24] Mobilyzer dashboard. https://openmobiledata.appspot.com.

[25] Mobilyzer data repository. https://console.developers.google.com/storage/openmobiledata_public/.

[26] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[27] Navigation Timing. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html.

[28] nDPI: Open and Extensible LGPLv3 Deep Packet Inspection Library. http://www.ntop.org/products/deep-packet-inspection/ndpi/.

[29] netem. https://wiki.linuxfoundation.org/networking/netem.

[30] Netflix: Internet connection speed recommendations. https://help.netflix.com/en/node/306.

[31] Netflix open connect. https://openconnect.netflix.com/en/.

[32] Number of apps available in leading app stores as of march 2017. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[33] Okhttp. http://square.github.io/okhttp/.

[34] Ookla NET INDEX EXPLORER. http://explorer.netindex.com/maps.

[35] Ooyala Global Video Index Q2 2014. http://go.ooyala.com/rs/OOYALA/images/Ooyala-Global-Video-Index-Q2-2014.pdf.

[36] OPTICOM, PESQ - perceptual evaluation of speech quality. http://www.opticom.de/technology/pesq.php.

[37] PhoneLab Testbed. http://www.phone-lab.org.

[38] The promise of quic: A faster, more flexible transport protocol. https://www.callstats.io/2017/06/02/quic/.

[39] QUIC Protocol. https://www.chromium.org/quic.

[40] Shadowsocks Socks5 Proxy. https://shadowsocks.org.

[41] SPDY Protocol – Draft 3.1. http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1.

[42] Streaming services now account for over 70netflix dominates with 37 https://venturebeat.com/2015/12/07/streaming-services-now-account-for-over-70-of-peak-traffic-in-north-america-netflix-dominates-with-37/.

[43] Telemetry. https://wiki.mozilla.org/Telemetry.

[44] The LTE Network Architecture, A comprehensive tutorial. Alcatel-Lucent.

[45] The Web App Manifest. https://developers.google.com/web/fundamentals/web-app-manifest/.

[46] University of Notre Dame NetSense Project: A study into the formation and evolution of social networks using mobile technology. http://netsense.nd.edu/.

[47] Web App Manifest. https://www.w3.org/TR/appmanifest/.

[48] WebPagetest - Website Performance and Optimization Test. http://www.webpagetest.org/.

[49] WebRTC Native Code. https://webrtc.org/native-code/.

[50] S. Agarwal and J. R. Lorch. Matchmaking for Online Games and Other Latency-Sensitive P2P Systems. In *Proc. of ACM SIGCOMM*, 2009.

[51] V. Aggarwal, E. Halepovic, J. Pang, S. Venkataraman, and H. Yan. Prometheus: Toward Quality-of-experience Estimation for Mobile Apps from Passive Network Measurements. In *Proc. of HotMobile*, 2014.

[52] H. F. Alan and J. Kaur. Can Android Applications Be Identified Using Only TCP/IP Headers of Their Launch Time Traffic? In *Proc. of WiSec*, 2016.

[53] Alcatel-Lucent. 9900 wireless network guardian, 2013. http://www.alcatel-lucent.com/products/9900-wireless-network-guardian.

[54] A. Auyoung, P. Buonadonna, B. N. Chun, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. *Two Auction-Based Resource Allocation Environments: Design and Experience*. John Wiley & Sons, Inc., 2009.

[55] P. Bahl, A. Adya, J. Padhye, and A. Walman. Reconsidering wireless systems with multiple radios. *ACM SIGCOMM Computer Communication Review*, 2004.

[56] F. Baker and G. Fairhurst. IETF Recommendations Regarding Active Queue Management . RFC 7567, Internet Engineering Task Force, 2015.

[57] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang. A Quest for an Internet Video Quality-of-experience Metric. In *Proc. of HotNets-XI*, 2013.

[58] M. Belshe, R. Peon, and E. M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, Internet Engineering Task Force, 2015.

[59] T. Bottger, F. Cuadrado, G. Tyson, I. Castro, and S. Uhlig. Open connect everywhere: A glimpse at the internet ecosystem through the lens of the netflix cdn. In *Proc. of ACM SIGCOMM Computer Communications Review (CCR)*, 2018.

[60] I. N. Bozkurt and T. Benson. Contextual Router: Advancing Experience Oriented Networking to the Home. In *Proc. of SOSR*, 2016.

[61] M. Butkiewicz, H. Madhyastha, and V. Sekar. Understanding Website Complexity: Measurements, Metrics, and Implications. In *Proc. of IMC*, 2011.

[62] I. Canadi, P. Barford, and J. Sommers. Revisiting Broadband Performance. In *Proc. of IMC*, 2012.

[63] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: A platform for educational cloud computing. In *Proc. of SIGCSE*, 2009.

[64] Carrier Compare. Compare speed across networks. `https://itunes.apple.com/us/app/carriercompare-compare-speed/id516075262?mt=8`.

[65] Carrier IQ. Carrier IQ: What it is, what it isn't, and what you need to know. `http://www.engadget.com/2011/12/01/carrier-iq-what-it-is-what-it-isnt-and-what-you-need-to/`.

[66] P. Casas, M. Seufert, and R. Schatz. YOUQMON: A System for On-line Monitoring of YouTube QoE in Operational 3G Networks. *SIGMETRICS Perform. Eval. Rev.*

[67] A. Chakraborty, S. Sanadhya, S. R. Das, D. Kim, and K.-H. Kim. ExBox: Experience Management Middlebox for Wireless Networks. In *Proc. of ACM CoNEXT*, 2016.

[68] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau. QoE Doctor: Diagnosing Mobile App QoE with Automated UI Control and Cross-layer Analysis. In *Proc. of IMC*, 2014.

[69] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone Energy Drain in the Wild: Analysis and Implications. In *Proc. of the ACM SIGMETRICS*, 2015.

[70] Y.-C. Chen, Y.-S. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley. A Measurement-based Study of MultiPath TCP Performance over Wireless Networks. In *Proc. of IMC*, 2013.

[71] Y.-C. Chen, D. Towsley, and R. Khalili. Msplayer: Multi-source and multi-path leveraged youtuber. In *Proc. of ACM CoNEXT*, 2014.

[72] Q. D. Coninck, M. Baerts, B. Hesmans, and O. Bonaventure. A first analysis of multipath tcp on smartphones. In *Proc. of PAM*, 2016.

[73] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *Proc. of SIGMOD*, 2003.

[74] A. Croitoru, D. Niculescu, and C. Raiciu. Towards WiFi Mobility without Fast Handover. In *Proc. of USENIX NSDI*, 2015.

[75] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafoo, K. Papagiannaki, and P. Steenkiste. The Cost of the "S" in HTTPS. In *Proc. of ACM CoNEXT*, 2014.

[76] Q. De Coninck and O. Bonaventure. Multipath QUIC: Design and Evaluation. In *Proc. of CoNEXT*, 2017.

[77] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or Both? Measuring Multi-homed Wireless Internet Performance. In *Proc. of IMC*, 2014.

[78] P. Deshpande, X. Hou, and S. R. Das. Performance Comparison of 3G and Metro-Scale WiFi for Vehicular Network Access. In *Proc. of IMC*, 2010.

[79] O. Dobrijevic, A. J. Kassler, L. Skorin-Kapov, and M. Matijasevic. Q-POINT: QoE-Driven Path Optimization Model for Multimedia Services. In *Proc. of WWIC*, 2014.

[80] F. Duchene and O. Bonaventure. Making Multipath TCP friendlier to Load Balancers and Anycast. In *Proc. of IEEE ICNP*, 2017.

[81] S. Egger, P. Reichl, T. Hoßfeld, and R. Schatz. "Time is bandwidth"? Narrowing the gap between subjective time perception and Quality of Experience. In *Proc. of IEEE ICC*, 2012.

[82] A. Elmokashfi, A. Kvalbein, J. Xiang, and K. R. Evensen. Characterizing delays in norwegian 3g networks. In *Proc. of PAM*, 2012.

[83] FCC Announces "Measuring Mobile America" Program. `http://www.fcc.gov/document/fcc-announces-measuring-mobile-america-program`.

[84] M. Fiedler, T. Hossfeld, and P. Tran-Gia. A generic quantitative relationship between quality of experience and quality of service. *IEEE Network*, 24(2):36–41, 2010.

[85] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, 2013.

[86] P. Georgopoulos, Y. Elkhatib, M. Broadbent, M. Mu, and N. Race. Towards Network-wide QoE Fairness Using Openflow-assisted Adaptive Video Streaming. In *Proc. of FhMN*, 2013.

[87] A. Gerber, J. Pang, O. Spatscheck, and S. Venkataraman. Speed Testing Without Speed Tests: Estimating Achievable Download Speed from Passive Measurements. In *Proc. of IMC*, 2010.

[88] M. Ghasemi, P. Kanuparthy, A. Mansy, T. Benson, and J. Rexford. Performance Characterization of a Commercial Video Streaming Service. In *Proc. of IMC*, 2016.

[89] I. Grigorik. Deciphering the Critical Rendering Path. `http://calendar.perfplanet.com/2012/deciphering-the-critical-rendering-path/`.

[90] Y. Guo, F. Qian, Q. A. Chen, Z. M. Mao, and S. Sen. Understanding On-device Bufferbloat for Cellular Upload. In *Proc. of IMC*, 2016.

[91] Y. E. Guo, A. Nikravesh, Z. M. Mao, F. Qian, and S. Sen. Accelerating Multipath Transport Through Balanced Subflow Completion. In *Proc. of MobiCom*, 2017.

[92] B. Han, F. Qian, S. Hao, and L. Ji. An Anatomy of Mobile Web Performance over Multipath TCP. In *Proc. of ACM CoNEXT*, 2015.

[93] B. Hesmans, G. Detal, S. Barre, R. Bauduin, and O. Bonaventure. SMAPP: Towards Smart Multipath TCP-enabled APPlication. In *Proc. of ACM CoNEXT*, 2015.

[94] B. Hesmans, H. Tran-Viet, R. Sadre, and O. Bonaventure. A first look at real Multipath TCP traffic. In *Proc. of International Workshop on Traffic Monitoring and Analysis*, 2015.

[95] C.-K. Hsieh, H. Falaki, N. Ramanathan, H. Tangmunarunkit, and D. Estrin. Performance evaluation of android ipc for continuous sensing applications. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2012.

[96] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *Proc. of SIGCOMM*, 2013.

[97] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proc. of MobiSys*, 2010.

[98] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proc. of ACM SIGCOMM*, 2014.

[99] N. Iya, N. Kuhn, F. Verdicchio, and G. Fairhurst. Analyzing the impact of bufferbloat on latency-sensitive applications. In *Proc. of IEEE ICC*, 2015.

[100] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Perfomance and Power Characteristics of 4G LTE Networks. In *Proc. of MobiSys*, 2012.

[101] K. Jang, M. Han, S. Cho, H. Ryu, J. Lee, Y. Lee, and S. Moon. 3G and 3.5 G Wireless Network Performance Measured from Moving Cars and High-Speed Trains. In *Proc. of Workshop on Mobile Internet through Cellular Networks*, 2009.

[102] J. Jiang, X. Liu, V. Sekar, I. Stoica, and H. Zhang. EONA: Experience-Oriented Network Architecture. In *Proc. of HotNets*, 2014.

[103] J. Jiang, V. Sekar, I. Stoica, and H. Zhang. Shedding Light on the Structure of Internet Video Quality Problems in the Wild. In *Proc. of CoNEXT*, 2013.

[104] J. Jiang, V. Sekar, and H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proc. of CoNEXT*, 2012.

[105] M. Jurvansuu, J. Prokkola, M. Hanski, and P. Perala. HSDPA Performance in Live Networks. In *Proc. of IEEE ICC*, 2007.

[106] M. Katsarakis, R. C. Teixeira, M. Papadopouli, and V. Christophides. Towards a Causal Analysis of Video QoE from Network and Application QoS. In *Proc. of Internet-QoE*, 2016.

[107] H. Kellerer, U. Pferschy, and D. Pisinger. *The Multiple-Choice Knapsack Problem*, pages 317–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[108] S. Khirman and P. Henriksen. Relationship between quality-of-service and quality-of-experience for public internet service. In *Proc. of PAM*, 2002.

[109] J. Kim, R. Khalili, A. Feldmann, Y. Chen, and D. Towsley. Multi-source multi-path HTTP (mhttp): A proposal. *CoRR*, abs/1310.2748, 2013.

[110] T. Kimura, M. Yokota, A. Matsumoto, K. Takeshita, T. Kawano, K. Sato, H. Yamamoto, T. Hayashi, K. Shiomoto, and K. Miyazaki. QUVE: QoE Maximizing Framework for Video-Streaming. *J. Sel. Topics Signal Processing*, 2017.

[111] P. Kortum, A. Rahmati, C. Shepard, C. Tossell, and L. Zhong. LiveLab: Measuring wireless networks and smartphone users in the field. `http://livelab.recg.rice.edu/traces.html`.

[112] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *Proc. of IMC*, 2010.

[113] S. S. Krishnan and R. K. Sitaraman. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-experimental Designs. In *Proc. of IMC*, 2012.

[114] S. S. Krishnan and R. K. Sitaraman. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-Experimental Designs. *IEEE/ACM Transactions on Networking*, 21(6):2001–2014, 2013.

[115] M. Laner, P. Svoboda, E. Hasenleithner, and M. Rupp. Dissecting 3G Uplink Delay by Measuring in an Operational HSPA Network. In *Proc. of PAM*, 2011.

[116] M. Laner, P. Svoboda, P. Romirer-Maierhofer, N. Nikaein, F. Ricciato, and M. Rupp. A comparison between one-way delays in operating HSPA and LTE networks. In *Proc. WINMEE*, 2012.

[117] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928, 1996.

[118] X. Liu, A. Sridharan, S. Machiraju, M. Seshadri, and H. Zang. Experiences in a 3G network: interplay between the wireless channel and applications. In *Proc. of MobiCom*, 2008.

[119] J. Martin and N. Feamster. User-driven Dynamic Traffic Prioritization for Home Networks. In *Proc. of W-MUST*, 2012.

[120] Measurement Lab website. `http://www.measurementlab.net/`.

[121] O. Mehani, R. Holz, S. Ferlin, and R. Boreli. An early look at multipath TCP deployment in the wild. In *Proc. of HotPlanet*, 2015.

[122] X. Mi, F. Qian, and X. Wang. SMig: Stream Migration Extension For HTTP/2. In *Proc. of ACM CoNEXT*, 2016.

[123] MySpeedTest App. `https://play.google.com/store/apps/details?id=com.num`.

[124] H. Nam, K.-H. Kim, and H. Schulzrinne. QoE Matters More Than QoS: Why People Stop Watching Cat Videos. In *Proc. of IEEE INFOCOM*, 2016.

[125] A. Nandugudi, A. Maiti, T. Ki, F. Bulut, M. Demirbas, T. Kosar, C. Qiao, S. Y. Ko, and G. Challen. PhoneLab: A Large Programmable Smartphone Testbed. In *Proc. of SENSEMINE*, 2013.

[126] NetRadar. `https://www.netradar.org/en/about`.

[127] C. Ng, P. Buonadonna, B. N. Chun, A. C. Snoeren, and A. Vahdat. Addressing strategic behavior in a deployed microeconomic resource allocator. In *Workshop on Economics of Peer-to-Peer Systems*, 2005.

[128] K. Nichols and V. Jacobson. Controlling Queue Delay. *Queue*, 2012.

[129] C. Nicutar, D. Niculescu, and C. Raiciu. Using Cooperation for Low Power Low Latency Cellular Connectivity. In *Proc. of ACM CoNEXT*, 2014.

[130] A. Nika, Y. Zhu, N. Ding, A. Jindal, Y. C. Hu, X. Zhou, B. Y. Zhao, and H. Zheng. Energy and Performance of Smartphone Radio Bundling in Outdoor Environments. In *Proc. of WWW*, 2015.

[131] A. Nikravesh, D. R. Choffnes, E. Katz-Bassett, Z. M. Mao, and M. Welsh. Mobile Network Performance from User Devices: A Longitudinal, Multidimensional Analysis. In *Proc. of PAM*, 2012.

[132] A. Nikravesh, Y. Guo, F. Qian, Z. M. Mao, and S. Sen. An In-depth Understanding of Multipath TCP on Mobile Devices: Measurement and System Design. In *Proc. of MobiCom*, 2016.

[133] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao. Mobilyzer: An Open Platform for Principled Mobile Network Measurements. Technical report, University of Michigan, 2014. `http://www.eecs.umich.edu/~zmao/Papers/Mobilyzer.pdf`.

[134] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao. Mobilyzer: An Open Platform for Controllable Mobile Network Measurements. In *Proc. of MobiSys*, 2015.

[135] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 2010.

[136] Ookla. Ookla speedtest mobile apps. http://www.speedtest.net/mobile/.

[137] P802.16.3 - standard for mobile broadband network performance measurements. http://standards.ieee.org/develop/project/802.16.3.html.

[138] C. Paasch, S. Barré, et al. Multipath TCP in the Linux Kernel. http://www.multipath-tcp.org.

[139] A. Patro, S. Rayanchu, M. Griepentrog, Y. Ma, and S. Banerjee. The anatomy of a large mobile massively multiplayer online game. In *Proc. of MobiGames*, 2012.

[140] Q. Peng, M. Chen, A. Walid, and S. Low. Energy Efficient Multipath TCP for Mobile Devices. In *Proc. of MobiHoc*, 2014.

[141] T. Pering, Y. Agarwal, R. Gupta, and R. Want. CoolSpots: Reducing the Power Consumption of Wireless Mobile Devices Using Multiple Radio Interfaces. In *Proc. of MobiSys*, 2006.

[142] PlanetLab website. http://www.planet-lab.org.

[143] Portolan. The portolan network sensing architecture. http://portolan.iet.unipi.it/.

[144] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger. TCP Revisited: A Fresh Look at TCP in the Wild. In *Proc. of IMC*, 2009.

[145] F. Qian, V. Gopalakrishnan, E. Halepovic, S. Sen, and O. Spatscheck. TM3: Flexible Transport-layer Multi-pipe Multiplexing Middlebox Without Head-of-line Blocking. In *Proc. of ACM CoNEXT*, 2015.

[146] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Profiling Resource Usage for Mobile Applications: a Cross-layer Approach. In *Proc. of MobiSys*, 2010.

[147] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. iMAP: Intelligent Mobile Application Profiling Tool. In *Proc. of MobiSys*, 2011.

[148] Qian, Feng and Sen, Subhabrata and Spatscheck, Oliver. Characterizing Resource Usage for Mobile Web Browsing. In *Proc. of MobiSys*, 2014.

[149] Qian, Feng and Wang, Zhaoguang and Gerber, Alexandre and Mao, Zhuoqing Morley and Sen, Subhabrata and Spatscheck, Oliver. Characterizing Radio Resource Allocation for 3G Networks. In *Proc. of IMC*, 2010.

[150] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proc. of ACM CoNEXT*, 2011.

[151] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proc. of ACM SIGCOMM*, 2011.

[152] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proc. of OSDI*, 2012.

[153] U. Reiter, K. Brunnström, K. De Moor, M.-C. Larabi, M. Pereira, A. Pinheiro, J. You, and A. Zgank. *Factors Influencing Quality of Experience*, pages 55–72. Springer International Publishing, 2014.

[154] RIPE NCC. Ripe atlas. `https://atlas.ripe.net/`.

[155] P. Romirer-Maierhofer, F. Ricciato, A. D'Alconzo, R. Franzan, and W. Karner. Network-Wide Measurements of TCP RTT in 3G. In *Proc. of TMA*, 2009.

[156] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks. In *Proc. of MobiCom*, 2014.

[157] M. A. Sánchez, J. S. Otto, Z. S. Bischof, D. R. Choffnes, F. E. Bustamante, B. Krishnamurthy, and W. Willinger. Dasu: pushing experiments to the internet's edge. In *Proc. of USENIX NSDI*, 2013.

[158] Sandvine. Global Internet Phenomena Report 1H 2014. `https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/1h-2014-global-internet-phenomena-report.pdf`, 2014.

[159] R. Schatz, T. Hoßfeld, and P. Casas. Passive YouTube QoE Monitoring for ISPs. In *Proc. of IMIS*, 2012.

[160] A. Schulman, V. Navday, R. Ramjeey, N. Spring, P. Deshpandez, C. Grunewald, and K. J. V. N. Padmanabhany. Bartendr: A practical approach to energy-aware cellular data scheduling. In *Proc. of MobiCom*, 2010.

[161] J. Sclamp and G. Carle. Measrdroid. `http://media.net.in.tum.de/videoarchive/SS13/ilab2/2013+05+02_1000+MeasrDroid/pub/slides.pdf`.

[162] S. Sen, J. Yoon, J. Hare, J. Ormont, and S. Banerjee. Can They Hear Me Now?: A Case for a Client-assisted Approach to Monitoring Wide-area Wireless Networks. In *Proc. of IMC*, 2011.

[163] I. Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE MultiMedia*, (4):62–67, 2011.

[164] J. Sommers and P. Barford. Cell vs. WiFi: on the performance of metro area mobile connections. In *Proc. of IMC*, 2012.

[165] H. Soroush, P. Gilbert, N. Banerjee, B. N. Levine, M. Corner, and L. Cox. Concurrent Wi-Fi for Mobile Users: Analysis and Measurements. In *Proc. of ACM CoNEXT*, 2011.

[166] Speedometer project source. `https://github.com/Mobiperf/Speedometer`.

[167] Speedtest.net. `http://www.speedtest.net/`.

[168] R. Stewart. Stream Control Transmission Protocol. RFC 4960, 2007.

[169] A.-J. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting behind Akamai: Travelocity-based detouring. In *Proc. of ACM SIGCOMM*, Pisa, Italy, September 2006.

[170] S. Sundaresan, S. Burnett, N. Feamster, and W. De Donato. BISmark: A testbed for deploying measurements and applications in broadband access networks. In *Proc. of USENIX ATC*, 2014.

[171] S. Sundaresan, W. De Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband Internet performance: A view from the gateway. In *Proc. of ACM SIGCOMM*, 2011.

[172] Y. sup Lim, Y.-C. Chen, E. M. Nahum, D. Towsley, R. J. Gibbens, and E. Cecchet. Design, Implementation and Evaluation of Energy-Aware Multi-Path TCP. In *Proc. of ACM CoNEXT*, 2015.

[173] W. L. Tan, F. Lam, and W. C. Lau. An Empirical Study on 3G Network Capacity and Performance. In *Proc. of IEEE INFOCOM*, 2007.

[174] Tobias Flach and Pavlos Papageorge and Andreas Terzis and Luis Pedrosa and Yuchung Cheng and Tayeb Karim and Ethan Katz-Bassett and Ramesh Govindan. An Internet-Wide Analysis of Traffic Policing. In *Proc. of ACM SIGCOMM*, 2016.

[175] F. Vacirca, F. Ricciato, and R. Pilz. Large-Scale RTT Measurements from an Operational UMTS/GPRS Network. In *Proc. of WICON*, 2005.

[176] P. Valerio. Using carrier wifi to offload iot networks. `http://goo.gl/aQ15ii`.

[177] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proc. of USENIX NSDI*, 2013.

[178] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proc. of USENIX NSDI*, 2014.

[179] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. of ACM SIGCOMM*, 2011.

[180] Q. Xu, A. Gerber, Z. M. Mao, and J. Pang. Acculoc: practical localization of performance measurements in 3g networks. In *Proc. of MobiSys*, 2011.

[181] Q. Xu, J. Huang, Z. Wang, F. Qian, A. Gerber, and Z. M. Mao. Cellular data network infrastructure characterization and implication on mobile content placement. In *Proc. of the ACM SIGMETRICS*, 2011.

[182] Q. Xu, Y. Liao, S. Miskovic, M. Baldi, Z. M. Mao, A. Nucci, and T. Andrews. Automatic Generation of Mobile App Signatures from Traffic Observations. In *Proc. of IEEE INFOCOM*, 2015.

[183] S. Xu, S. Sen, Z. M. Mao, and Y. Jia. Dissecting VOD Services for Cellular: Performance, Root Causes and Best Practices. In *Proc. of IMC*, 2017.

[184] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. Investigating Transparent Web Proxies in Cellular Networks. In *PAM*, 2015.

[185] Y. Xu, C. Yu, J. Li, and Y. Liu. Video Telephony for End-consumers: Measurement Study of Google+, iChat, and Skype. In *Proc. of IMC*, 2012.

[186] H. Yao, G. Ranjan, A. Tongaonkar, Y. Liao, and Z. M. Mao. SAMPLES: Self Adaptive Mining of Persistent Lexical Snippets for Classifying Mobile Application Traffic. In *Proc. of MobiCom*, 2015.

[187] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *SIGCOMM*, 2015.

[188] K. Zarifis, T. Flach, S. Nori, D. Choffnes, R. Govindan, E. Katz-Bassett, Z. M. Mao, and M. Welsh. Diagnosing path inflation of mobile client traffic. In *Proc. of PAM*, 2014.

[189] X. Zhang, Y. Xu, H. Hu, Y. Liu, Z. Guo, and Y. Wang. Profiling Skype video calls: Rate control and video quality. In *Proc. of INFOCOM*, 2012.

[190] H. Zheng, E. K. Lua, M. Pias, and T. G. Griffin. Internet routing policies and round-trip-times. In *Proc. of PAM*, 2005.